Fairness and Liveness under Weak Consistency

Parosh Aziz Abdulla[1][0000−0001−6832−6611]✉, Mohamed Faouzi Atig[1][0000−0001−8229−3481], Adwait Godbole[2][0000−0001−7704−304X], Shankaranarayanan Krishna[3][0000−0003−0925−398X], and Mihir Vahanwala[4][0009−0008−5709−899X]

[1] Uppsala University, Uppsala, Sweden {parosh,mohamed_faouzi.atig}@it.uu.se
[2] University of California Berkeley, Berkeley, USA adwait@berkeley.edu
[3] IIT Bombay, Mumbai, India krishnas@cse.iitb.ac.in
[4] Max Planck Institute for Software Systems, Saarland Informatics Campus, Saarbrcken, Germany mvahanwa@mpi-sws.org

Abstract. We consider the verification of concurrent programs running on weakly consistent platforms, i.e., weaker semantics than the classical Sequential Consistency (SC) semantics. We describe a framework for the verification of liveness properties for such programs. To that end, we introduce a notion of fairness that combines the classical transition fairness condition with an additional condition that forbids demonic behaviors of the memory system. We illustrate the framework by instantiating it for the classical Total Store Order (TSO) memory model. The presentation is tutorial-like and based on our previous works [3, 4].

1 Introduction

The ubiquity of parallel systems has resulted in an extensive research effort to increase their efficiency, security, and reliability. While designing concurrent systems has always been a difficult challenge, developing correct concurrent systems has become even more challenging in recent years. The main reason is that computing platforms do not provide the fundamental guarantee of *Sequential Consistency* (SC) anymore. The SC semantics interleaves the parallel executions of different processes while preserving the order of actions performed by a single process [19]. The SC model is easy to understand since all components are strongly synchronized so that they all have a *uniform view* of the global state of the system. The SC semantics is so intuitive that programmers of concurrent applications often assume that it is guaranteed by the platforms on which they run their applications. On the flip side, SC is too expensive to maintain since it requires continuous synchronization of all system components. At the hardware level, SC requires strong coherence; at the distributed system level, SC requires that updates by a given site are immediately conveyed to all other sites. Such strong consistency requirements are impossible to achieve with reasonable efficiency and energy consumption. Therefore, modern platforms implement optimizations that lead to the relaxation of the inter-component synchronization, offering only *weak* semantic guarantees. The problem is that program behaviors may differ considerably from their behaviors under the SC semantics when run on such platforms. Even standard textbook programs may exhibit faulty behaviors when run under weakly consistent semantics. This paper will describe the classical Dekker mutual exclusion protocol as a case in point. The discrepancy in program behaviors gives rise to new challenges to maintaining the reliability and security of concurrent applications. Extensive research has been undertaken to answer the challenges of modeling, testing, and verifying applications running according to such semantics.

Historically, two classes of specifications have been prominent in program verification, namely safety and liveness properties. Roughly speaking, a safety property states that "nothing bad will happen during the execution of the program" and a liveness property states that "something good will happen during the execution of the program." [16, 20]. Despite these properties being complementary, verification frameworks for liveness are usually more complicated. First, checking safety properties, in many cases, can be reduced to the (simple) reachability problem, while checking liveness properties usually amounts to checking repeated reachability of states [22]. Second, concurrency comes with an inherent *scheduling non-determinism*, i.e., at each step, the scheduler may non-deterministically select the following process to run. Therefore, liveness properties must be accompanied by appropriate fairness conditions on the scheduling policies to prohibit trivial blocking behaviors [20]. In the example of two processes trying to acquire a lock, demonic non-determinism [15] may always favor one process over the other, leading to starvation.

The verification of liveness properties (and also safety properties) has attracted much research in the context of programs running under the classical Sequential Consistency (SC) [19]. There is a clear gap in weakly consistent semantics: most existing works concentrate on safety properties, and only recently have works started to appear on verifying liveness properties.

This paper describes a framework introduced in [4] for verifying liveness properties under weak memory models. In Section 2, we will recall general concepts related to verification, concurrency, and weak memory models, and then describe their instantiation to the SC case in Section 3. We illustrate the challenges that arise in the case of weak memory through the TSO semantics. We recall the model in Section 4 and then study fairness and liveness properties for TSO in Section 5. In particular, we show the equivalence of memory fairness and probabilistic fairness. In Section 6, we consider the verification of Markov chains induced by the probabilistic model of TSO. In Section 7, we give some conclusions and directions for future work.

2 Concurrency, Shared Memory, and Verification

A concurrent program consists of multiple processes, or threads, whose executions depend on their scheduling, and communication via shared memory. Processes are defined by the instructions they must execute; they have access to local variables, or registers, in order to do so. The control state refers to the position of the instruction pointers and the contents of the registers of each of the processes. The shared memory consists of global variables, or locations, that can be addressed by all processes.

For the sake of intuition, it might be helpful to anthropomorphize the processes and think of the shared memory as a messaging service that operates according to the protocol governed by the memory model. Processes interface with the shared memory through write and read operations: these respectively correspond to sending and fetching messages.

The verification problem for a fixed memory model asks, does the control state satisfy the given specification along the run of the concurrent program? For instance, when there is a critical shared resource, it is natural to ask whether it will never be accessed simultaneously by two processes (mutual exclusion, an example of safety) and whether each process is guaranteed to access it often enough (an example of liveness). The answer to such synchronization decision problems crucially depends on the underlying communication protocol, i.e. the memory model.

As mentioned in the Introduction, safety properties specify that "bad things never happen". In the above example, in order to verify safety, the intuitive idea is to prove (by the absence of a counterexample) that there is never a fatal miscommunication. On the other hand, liveness, which specifies that "good things are guaranteed to eventually happen," is harder to meaningfully verify because it often requires that the pivotal message delivery (and scheduling) be *consistently* good: a requirement that is seldom built into the protocols we consider. Therefore, we use judicious *fairness conditions* to rule out behaviors that the protocol technically permits but are practically unreasonable in the long term.

In this paper, we shall consider control state reachability as our canonical safety problem, and termination and repeated state reachability as our canonical liveness problems.

We shall now give an intuitive overview of the key aspects of some fundamental (weak) memory models, and motivate our notions of fairness. As we shall see later, it turns out that under these fairness conditions, our liveness problems reduce to safety queries. We will rely on the intuition of anthropomorphic processes using the shared memory as a messaging service (writes and reads respectively correspond to sending and fetching messages) in the following subsection.

2.1 Memory Models: Intuition

Processes, as users of the messaging system, can only send or request messages through an interface that is common for all protocols. We visualize each process to have a separate mailbox for messages pertaining to each shared variable. To send a message (i.e. write value v to variable x), a process submits it to the messaging service, which then assumes the responsibility of annotating the message with relevant metadata and delivering it to the other processes' x-mailboxes. At the other end, to fetch a message (i.e. to read from variable x), a process requests a message from its x-mailbox, and the service grants one as per the messaging protocol. We assume that on a given day, the service handles at most one write or read.

Sequential Consistency (SC). This is the simplest protocol to reason about. Whenever a process sends a message (writes value v to variable x), the service broadcasts and delivers it *immediately* to all processes' x-mailboxes. When a process requests to fetch a message (read from variable x), the service gives it the most recent message from the mailbox. In this protocol, one can see that the service can discard all but the most recent messages to every variable.

TSO and PSO. As one would expect, it is costly and demanding to implement the high degree of synchronization described above. A popular paradigm is that of *multi-copy atomicity*, exemplified by Total Store Order (TSO) and Partial Store Order (PSO) [21, Section 8 and Appendix D] (see also [14]) - it provides performance by weakening SC's consistency guarantees, while also keeping the protocol centralized enough for its analysis to be manageable.

Total Store Order (TSO). This protocol is not as prompt as SC, but similar in most aspects. The service maintains a buffer, or a FIFO queue, for the outgoing messages of each process. When a process p sends a message (writes value v to variable x), it is added to p's mailbox immediately, but the broadcast is delayed: the service adds it to the queue. To perform its broadcasting duties, the service picks a queue, takes the message at the front, and delivers it to the appropriate mailboxes of all the other processes. When a process p requests to fetch a message (read from variable x), the service first checks p's queue for an x-message and returns the most recent one. If there aren't any, the service returns the most recent message from the mailbox.

Partial Store Order (PSO). PSO differs from TSO by allowing writes to race, i.e. for each process, the service maintains a separate queue for every variable. All other aspects of the protocol are the same. Once again, we observe that the service need only keep track of the messages in the queues and most recently broadcast messages of each variable.

SRA and WRA. As parallel systems get more distributed, the simplicity of multi-copy atomicity no longer offsets the cost of simultaneous broadcasting. In these settings, causal consistency, exemplified by Strong Release Acquire (SRA) and Weak Release Acquire (WRA) [18, Sections 3-4] is the paradigm of choice. The idea is that reading messages creates causal dependencies which must be respected: let Alice write two messages to x, and Bob write a message to y after reading them. Then, if Eve reads Bob's message, the service is forbidden from giving Eve Alice's first message should she wish to read from x - in a certain sense, this is causally far before the events Eve is already aware of, and too stale an update.

Strong Release Acquire (SRA). The messaging service maintains a clock for each variable, and a vector of timestamps for each process, with an entry corresponding to each clock. The clocks are used to totally order all writes to the same variable, the timestamps determine what messages have not been rendered redundant to a process by causal dependencies. When process p sends a message (writes value v to variable x): (i) The x-clock is incremented by 1; (ii) The x-clock entry in p's timestamp vector is updated to the current value; and (iii) The message is annotated with p's new timestamp vector. This message is delivered immediately to p's mailbox, but to other processes, the service can deliver it asynchronously and at leisure.

On the other hand, when p requests a message (reads from x), (i) the service selects a message from p's mailbox whose x-timestamp annotation is at least as large as the x-clock entry in p's vector of timestamps; (ii) the service hands this message, which is guaranteed by construction to not be redundant, over; and (iii) finally, to enforce new dependencies that are created by this read, for each clock, the service updates the corresponding entry in p's vector of timestamps to be the maximum (a) of the old value and (b) the timestamp in the annotation of the message.

Weak Release Acquire (WRA). The protocol is the same with respect to the propagation of messages and updates of timestamps and clocks. The key difference that makes this model significantly weaker than SRA is that the service maintains a clock for every (variable, process) pair. Hence, timestamp vectors also have entries corresponding to each (variable, process) pair.

In these protocols too, the service can safely discard messages which it cannot give to any process for reading: these are messages whose timestamps are smaller than the timestamps of every process.

Remark: Other Models. In the multi-copy atomic models we considered, reads had *acquire* semantics: they were not allowed to race. Relaxed Memory Ordering (RMO) [21, Section 8 and Appendix D] is a multi-copy atomic model that weakens PSO by allowing reads to race if they are sufficiently independent of each other, and/or the writes they overtake. In such a protocol, the service would be allowed to buffer read requests too, instead of being obliged to address them immediately.

FIFO consistency [13] is a memory model that is neither multi-copy atomic nor causally consistent. Like TSO, the messaging service maintains for each process, a queue of its writes. However, unlike TSO, these writes are propagated to other processes asynchronously, rather in the style of SRA. Unlike SRA though, causal dependencies are not tracked. When a process wishes to read variable x, the service hands it the most recent x-message that was delivered to it.

2.2 Memory Fairness: Intuition

A common observation for all weak memory models is that the messaging service, i.e., memory, needs to store only those messages that are under propagation and/or have not been rendered redundant by the protocol yet. If the service is an *efficient* medium of communication, the number of such messages will be minimal. In the extreme case, consider SC: it is the most efficient protocol, and at any point, it only needs to store one message per variable.

We refer to the number of messages the service is keeping track of (in models like RMO, this also includes buffered reads) as the *size* of the memory configuration. Configurations with minimal size (one message per variable, no pending reads) are called *plain*.

Recall the requirements for liveness properties we had outlined earlier: consistently good communication between processes, facilitated by consistently efficient message delivery. The latter is what we intended to capture with fairness conditions. The notion of configuration size allows us to accurately quantify the efficiency of messaging, which is inversely related to the extent of weak behavior. Hence, our plan will be to devise formal definitions of fairness that imply, for example, that the configuration size is bounded, or that plain configurations are visited infinitely often: such are the hallmarks of a well-functioning messaging system.

In the following sections, we consider SC and TSO and use running examples to examine the above concepts of shared memory, verification, and fairness notions in more detail. We remark that one can indeed use the intuition of shared memory as a medium of communication and notion of configuration size based fairness described in this section to adapt the following technical discussion to other memory models.

3 Sequential Consistency (SC)

We will first recall the SC semantics and describe the verification of safety properties. Then, we consider liveness properties and describe different fairness conditions needed to verify such properties.

3.1 Model

To illustrate our framework, we consider a simple model for concurrent programs, consisting of processes that communicate by performing read (load) and write (store) operations on a set of shared (global) variables. Fig. 1 depicts such a program with two processes p_1 and p_2 sharing two shared variables x and y. In this section, we consider the SC semantics for such programs where process operations are atomic. When the process p_1 performs the write operation, assigning the value 3 to the variable x, we simultaneously update the value of x in the memory and, hence, the new value of x will be immediately readable by p_2. When p_2 reads the value of x in the second transition, it sees the latest value written to x, namely 3.

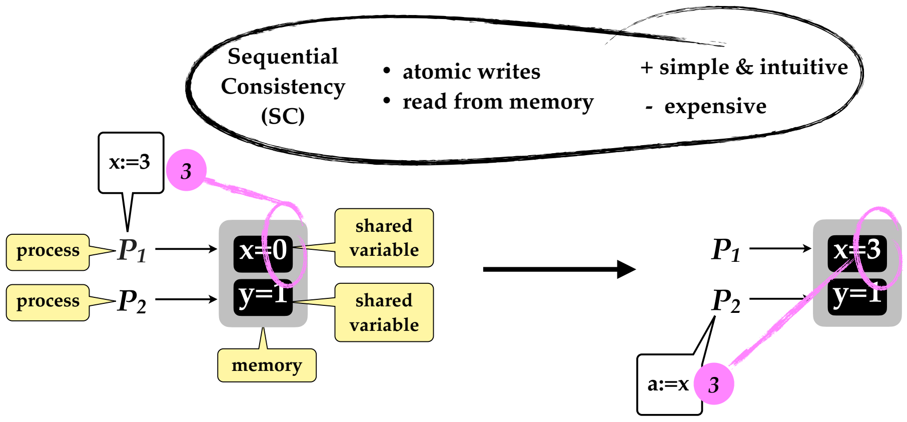


Fig. 1. The SC semantics.

3.2 Safety Properties

The classical interpretation of a safety property is that "nothing bad will happen during the program's execution" [16, 20]. This means that we can reduce checking safety properties to the reachability of a set of *bad* configurations that violate a given requirement of the program. A typical example is *mutual exclusion* where concurrent processes contend a shared resource. The bad configurations are those in which multiple processes access the shared resource. We depict the classical Dekker mutual exclusion protocol in Fig. 2. The program contains two processes p_1 and p_2 sharing two variables x and y. The processes have a local variable each, namely a resp. b. The goal of the protocol is to guarantee *mutual exclusion*, i.e., to prevent the processes from entering their critical section, $\texttt{CS}_1$ and $\texttt{CS}_2$, simultaneously. Before moving to its critical section, the process p_1 declares its intention by setting the shared variable x to 1, and then reads y's value, storing it in its local variable a. It halts its progress to its critical section if the value of a is equal to one, i.e., if p_2 is about to enter, or it is inside its critical section. The process p_2 uses the same scenario to synchronize with p_1. Under the SC semantics, the above scenario succeeds, and the program will satisfy mutual exclusion. To see this, assume, without loss of generality, p_1 executes the first instruction. Since the transition assigns 1 to x, process p_2 assigns the value 1 to its local variable b when it executes its read instruction, and hence it will not be able to cross to its critical section.

3.3 Fairness and Liveness

A liveness property states that "something *good* will eventually happen" during the program's execution. For the Dekker protocol, a typical liveness property is that each process infinitely often enters its critical section during any program

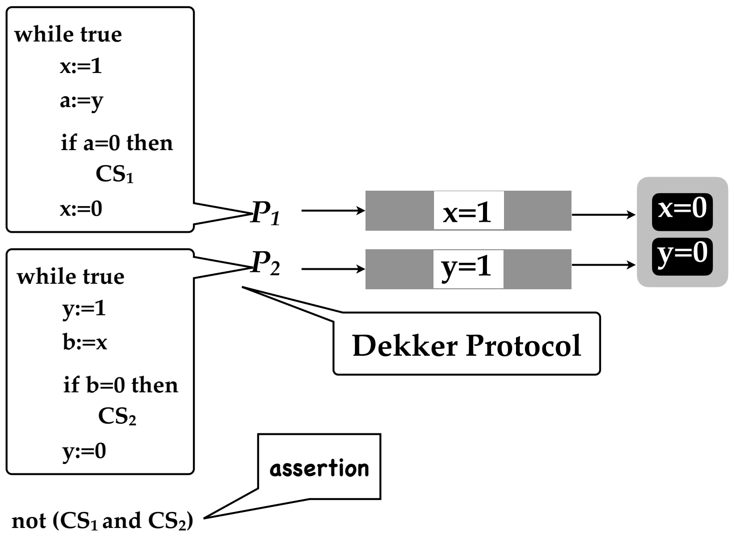


Fig. 2. The Dekker Protocol

run. Liveness properties can be trivially violated without some *fairness condition* that provide certain guarantees concerning the process scheduling. In the case of Dekker, a run that never schedules p_2 to run will trivially violate the above liveness condition since p_2 will never reach its critical section. The literature contains many fairness conditions for verifying liveness properties (see, e.g., [16, 20].). The classical *strong fairness* condition states that we must schedule each process infinitely often along each program run. In the Dekker example, we schedule both p_1 and p_2 infinitely often. While strong fairness ensures that p_2 is scheduled infinitely often, it still fails to guarantee that p_2 reaches its critical section infinitely often (or even once). This can happen if p_2 is always scheduled to run when $x = 1$. Although p_2 will perform infinitely many iterations of its loop, it will always fail the if-statement guarding the critical section. A more robust and helpful notion of fairness is *transition fairness* (called strong transition fairness in the comprehensive [17, Sections 4-5]): if a transition t in a process is enabled infinitely often, then t will be taken infinitely often. Transition fairness prevents the above demonic behavior and guarantees that both p_1 and p_2 in the Dekker protocol visit their critical sections infinitely often.

As another example, consider the simple program of Fig. 3. We want to verify that process p_2 always terminates. Strong fairness is not enough to prove p_2's termination. The following strongly fair but demonic scheduling policy prevents p_2 from ever terminating:

- p_1 assigns 1 to the shared variable x.
- p_2 assigns 2 to the shared variable x.
- p_2 reads the value 2 of x and stores it in its local variable a.

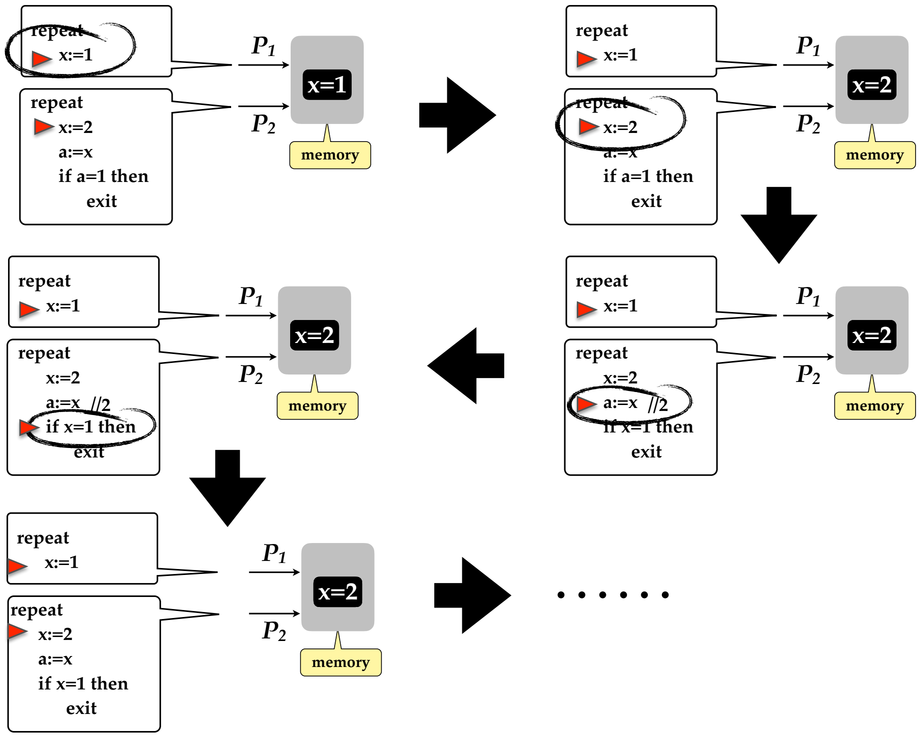


Fig. 3. Simple Program.

- p_2 fails the condition of the if-statement and re-starts the while-loop.
- p_1 assigns 1 to the shared variable x, and we repeat the above sequence.

However, p_2 is guaranteed to terminate under transition fairness since p_1 is guaranteed to be scheduled after p_2 has executed the instruction $x := 2$. Hence, the value of a will be one when p_2 reaches its if-statement.

4 Total Store Order (TSO)

We consider one of the most fundamental weak memory models: the Total Store Order (TSO) semantics. We depict the semantics in Fig. 4 for the case where we have two processes p_1 and p_2 sharing two variables x and y. The operational semantics of TSO make write instructions non-atomic by inserting an unbounded FIFO buffer between each process and the globally visible memory. Technically, the memory comprises both the process buffers and the globally visible component. However, in the context of TSO, we will refer to the latter as "main memory" or simply "memory" for convenience. We refer to the buffers as the *store buffer*, or the *pending buffer* of p_1 resp. p_2. Fig. 5 illustrates a typical program run, starting from a configuration γ_0 where the store buffers are empty,

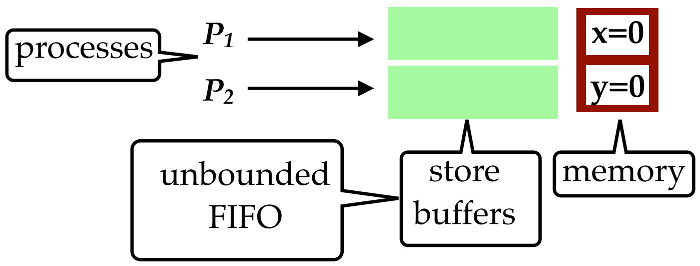


Fig. 4. The TSO architecture.

and the memory state is $x = 0$ and $y = 0$. To simplify the presentation, we omit the local states of the processes p_1 and p_2 and only depict the store buffers and the memory. In the first step, the process p_1 writes the value 1 to the variable x. Instead of immediately updating the memory, p_1 appends a *write message* $x = 1$ to the end of its store buffer, obtaining the configuration γ_1. From γ_1, the process p_1 writes the value 2 to the variable x, so it appends the message $x = 2$ to the store buffer. In γ_2, the store buffer of p_1 in γ_2 contains two messages, while the store buffer of p_2 is still empty. Next, p_1 tries to read the value of x. To that end, it checks its store buffer. If the buffer contains a message on x, then p_1 fetches the value of the latest such a value (2, in the case of γ_2). We say that p_1 performs a *read-own-write* operation since it reads a write instruction it has issued. In γ_3, the process p_1 reads the value of the variable y. Since there are no pending write messages on y in the buffer of p_1 in γ_3, it fetches the value 0 of y from the memory. We all this a *read-from-memory* operation. We observe that read operations do not change the states of the pending buffers or the memory. In a similar manner, p_2 reads the value of y in γ_4. Although p_1 has already performed the write operations on x, their effects are invisible to p_2 since the corresponding message has still not reached the memory. From γ_5, the program performs an *update* operation, where it takes the message $x = 1$ at the *head* of the buffer of p_1 and uses it to update the value of x in the memory. This value will now be visible to p_2 in γ_6. In a nutshell, the TSO semantics is defined by: (i) writing-to-store-buffer, (ii) reading-own-writes. (iii) reading-from-memory. (iv) updating the memory.

The TSO semantics is weaker than the SC semantics. In particular, TSO can mimic any SC run by performing an update after each write instruction. On the other hand, TSO allows more behaviors, introducing bugs in programs that are correct under SC. The Dekker protocol of Section 3.1 is a case in point. We will use the run of Fig. 6 to explain why the Dekker protocol does not satisfy mutual exclusion under the TSO semantics. Recall that, before entering its critical section, a process performs a write operation that prevents the other process from moving towards its critical section. Since write operations are not

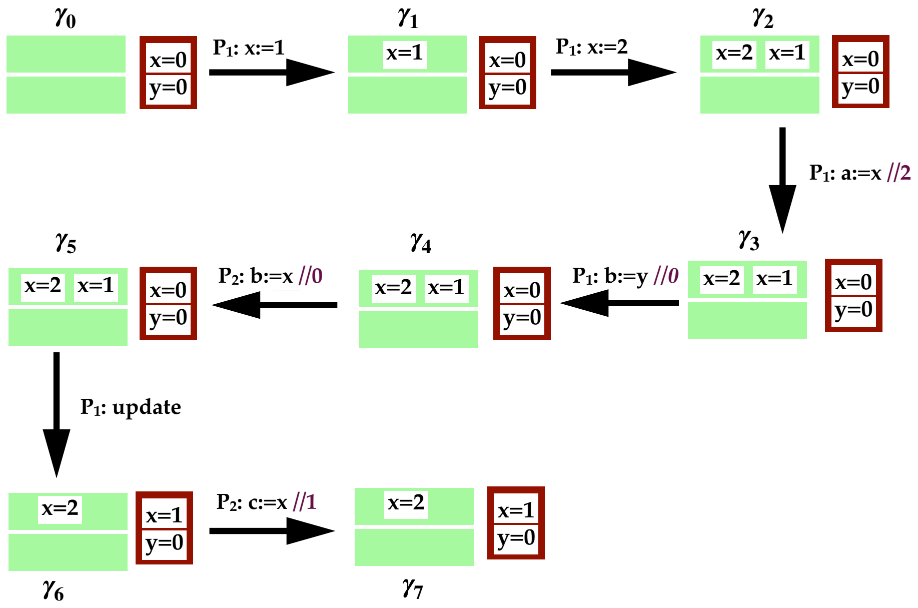


Fig. 5. A run according to the TSO semantics

atomic in the case of TSO, the above scenario will fail, and both processes can move to their critical sections simultaneously. Concretely, the processes p_1 and p_2 perform their write operations appending the corresponding messages to their buffers (the configurations γ_2 and γ_3). When p_1 performs its read operation from γ_3, it does not find pending write messages on y in its buffer, and hence it fetches the value of y from the memory. Since the message on y in process p_2's buffer has not reached the memory, it is not visible to p_1, and p_1 will read the value $y = 0$. Analogously, p_2 reads the value $x = 0$; hence, both processes can enter their critical sections.

5 Fairness and Liveness under TSO

5.1 Transition Fairness

While transition fairness allows the verification of liveness properties for a wide range of concurrent programs under SC, it is too weak in the case of TSO. In Fig. 7, we re-consider the program of Fig. 3 and run it under TSO. We show that, in contrast to SC, there is now a transition-fair run in which P_2 does not terminate. Roughly speaking, transition fairness implies, among other things, that memory updates occur infinitely often, but it allows updates to occur less frequently than write instructions. This means that we will have program executions in which the process buffers always contain messages and never see

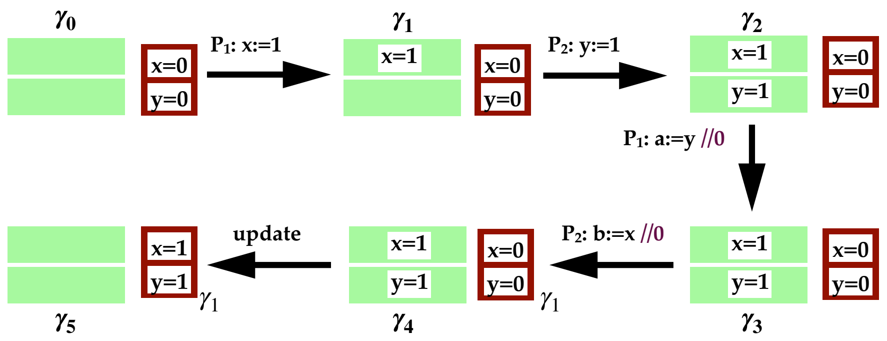


Fig. 6. A run of the Dekker Protocol under TSO semantics

the other processes' write operations. We start from the configuration γ_0 where both buffers are empty. The processes p_1 and p_2 execute their write instructions, and we obtain γ_2. In γ_2, the process p_2 reads the value of x. It sees the value of 2 from its buffer, so it will not terminate at this stage. In γ_3, the process p_1 moves the message in its buffer to update the value of x in the memory to 1. Although the value of x in the memory is 1, which is what p_2 needs to terminate, the latter fails to terminate in γ_4 since it still has the message $x = 2$ in its buffer. In the rest of the configurations, this scenario is repeated. Although transition fairness means that the value of x will be equal to 1 infinitely often. The messages from the buffer of p_2 are transferred to the memory infinitely often, p_2's buffer will never become empty. Hence, p_2 will never see $x = 1$ in the memory, meaning it will not terminate.

5.2 Memory-Boundedness Transition (MBT) Fairness

As we observed in the previous sub-section, the problem with transition fairness is that it allows runs in which the store buffers never become empty. The non-empty buffers confine the processes to only reading their own writes, effectively preventing inter-process communication through the shared memory. Therefore, we introduce a new notion of fairness, called *Memory-Boundedness (MB) fairness* that we will use together with transition fairness to strengthen the latter. MB-fairness means that we assume the existence of an (unknown) upper bound on the number of messages that can reside inside the store buffers at any point during the program runs. It is essential to notice that the upper bound is not given and can be arbitrarily large; MB-fairness assumes its mere existence. We use *MBT-fairness* to refer to the conjunction of MB-fairness and transition fairness. MBT-fairness enjoys two properties that make it attractive in the verification of liveness properties under weak memory models, namely (i) *modeling*: it eliminates the demonic behaviors that arise due to bad scheduling and buffer clogging; (ii) it allows algorithmic verification. We will consider the

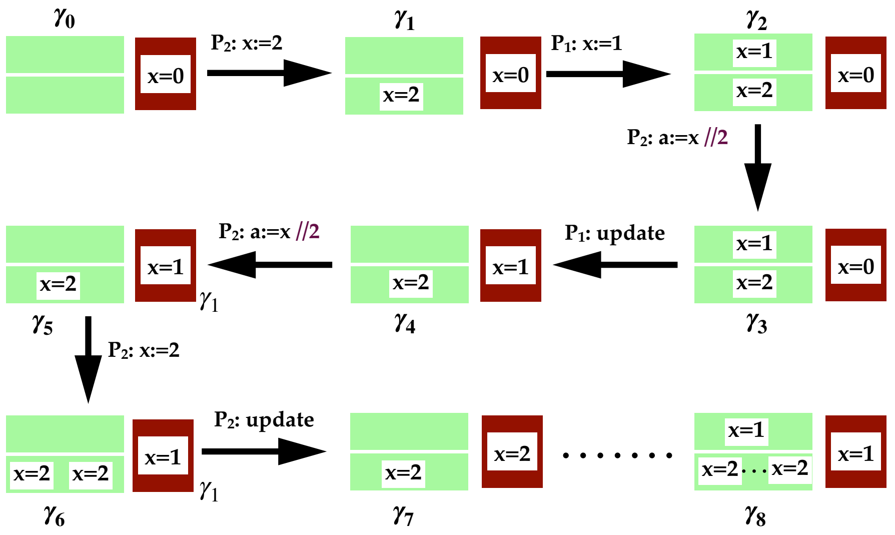


Fig. 7. A transition-fair run under the TSO semantics

modeling aspect in the rest of this section and discuss the verification aspect in Section 6.

Let us study the behavior of the program of Fig. 3 under TSO and MBT-fairness. Assume that the bound on the buffer size is (an unknown value) b. Consider an (infinite) run ρ of the program. By MBT-fairness, all the configurations have buffers of size at most b. Since the update operations are always enabled, it follows by transition fairness that there are infinitely many configurations along ρ of size at most $b-1$. By repeating the reasoning, there are infinitely many configurations along ρ with empty buffers. For such configurations, the program essentially behaves like in the case of SC. Using the same reasoning as in Section 3.3, we conclude the p_2 will terminate.

5.3 Probabilistic Memory Fairness

The notion of MBT-Fairness is arguably dependent on specific aspects of our formalism of the memory. However, as observed in the previous paragraph, it has a crucial consequence that can be expressed in terms of more universal notions, viz. that plain configurations are necessarily visited infinitely often. This property will be the backbone of the verification techniques we adopt.

In this subsection, we illustrate a natural alternative way of ensuring the above property, namely, by viewing the underlying transition system from a stochastic perspective. Interpreting transition systems as Markov Chains yields not only insights on the phenomena being modeled but also a rich body of algorithms to *quantitatively* understand their properties. Thus, we reinforce the benefits of our fairness notions on both the modeling and algorithmic fronts.

The transition system modeling our concurrent program can be converted into a Markov Chain by adding probabilities to the transitions: if there is a transition from configuration γ to configuration γ', then it is assigned a nonzero probability. Probabilities of transitions originating at γ must sum up to 1. By construction, an infinite run of a Markov Chain is transition fair with probability 1.

Probabilistic Memory Fairness. A Markov Chain representing a concurrent program satisfies Probabilistic Memory Fairness if an infinite run visits the set of plain configurations (i.e. those with empty buffers) infinitely often with probability 1.

5.4 Equivalence of Fairness Notions

We conclude this section by demonstrating the sense in which the two notions of memory fairness we introduced are equivalent. This will help us appreciate and leverage the algorithmic techniques presented in the next section.

Recall that our canonical liveness problems are repeated control state reachability and termination. In this subsection, we will consider the former: it is straightforward to adapt the arguments to the latter.

Equivalence Property. There exists N such that for all $n \geq N$, all transition fair runs with configuration size bounded by n visit control state c infinitely often if and only if c is visited infinitely often with probability 1 under probabilistic memory fairness.

Justification. We will demonstrate the equivalence by showing a canonical procedure that decides both repeated reachability queries. Consider a (colored) graph $G(n)$ parametrized by n. The vertices are the finitely many plain configurations (there are finitely many variables, and we assume that they can take finitely many values; in the case of TSO, these are configurations with empty buffers). We draw an edge from γ to γ' if and only if γ' is reachable from γ via configurations of size at most n. Furthermore, nodes are colored green or black. A node γ is colored green if and only if control state c is reachable from γ via configurations of size at most n.

By construction, it is clear that for all n, strongly connected components (SCCs) of the graph $G(n)$ will be uniformly colored. Moreover, this is a finite graph, and by construction, as n increases, edges can only be added, and vertices can only go from black to green. Thus, the graph will saturate at some $n = N$, and be the same thereafter. Let this saturated graph be G, and consider $n \geq N$.

Note that for G, the vertices are plain configurations, the edges and colors indicate reachability *without* restrictions on the size of intermediate configurations. A crucial remark is that G can be computed with conventional reachability queries. This G will serve as our canonical construct.

Recall that any transition fair run with configuration size bounded by n necessarily visits plain configurations infinitely often. Further, by transition fairness, it is guaranteed to eventually sink into a bottom SCC of $G(n) = G$. By similar reasoning, one can also argue that under probabilistic memory fairness, a run eventually sinks into a bottom SCC and visits plain configurations infinitely often with probability 1.

We are now ready to argue that the following three statements are equivalent: (i) all transition fair runs with configuration size bounded by n visit c infinitely often, and (ii) a run under probabilistic memory fairness visits c infinitely often with probability 1, and (iii) all reachable bottom SCC's (SCC's with no outgoing edges) of G are colored green.

The existence of a black reachable bottom SCC indicates a path where, after a finite prefix of nonzero probability, c can never be visited again.

For the converse, recall that if a transition fair run visits a state γ infinitely often, then it necessarily also visits each state γ' reachable from γ infinitely often. Similarly, if a run under probabilistic fairness visits a state γ infinitely often, then under probabilistic memory fairness, each state γ' reachable from γ is also visited infinitely often with probability 1. We apply these observations to the finite set of green nodes which is visited infinitely often, and from which c is reachable, and we are done.

The Centrality of Reachability. As a corollary of the above justification, we note that the core subroutines driving the algorithm deciding repeated reachability (liveness) under our fairness conditions are *simple* reachability (safety) queries. The construction of the canonical graph $G(N) = G$ and indeed, the computation of N reduces to safety queries. Established techniques to resolve them, along with standard algorithms to study the special class of Markov Chains we use, yield a solution to the problem of liveness verification.

6 Verification of Markov Chains

Having studied declarative formulations of fairness notions from the modeling and stochastic perspectives, we now complete the picture by discussing the *algorithmic* perspective. In particular, we consider the stochastic setting (the concurrent program induces a Markov Chain), where fairness is imposed through conditions on the probabilities assigned to the transitions. We will take the TSO memory model as a concrete example: one can use the intuition in Section 2 to adapt the observations to other memory models too.

We have already observed how deciding the reachability of a given set F plays a central role in verification. In the context of safety, F is a set of fatal configurations: safety is disproven by showing a (necessarily finite) path to F; it is typically proven by giving an *invariant*, i.e. a subset G of the set of configurations $\widetilde{F}$ from which F is not reachable. We will assume that for any configuration γ, we can decide whether $\gamma \in F$, and whether $\gamma \in \widetilde{F}$.

In the context of liveness, on the other hand, reaching F constitutes desirable behavior that one would want to guarantee or, in the setting of Markov Chains, quantify. More specifically, a decision problem could ask whether F is reached with probability $p \pm \varepsilon$, i.e. whether $|Prob_{\gamma_0}[\Diamond F] - p| < \varepsilon$.

The Algorithm. Given that we can detect both when a configuration γ is in F, and when it is in $\widetilde{F}$, the following approach might immediately spring to mind: maintain an under-approximation α_- (initialized to 0) and an over-approximation α_+ (initialized to 1) of $Prob_{\gamma_0}[\Diamond F]$. Starting from the initial configuration, generate all runs through the transition system in a breadth-first manner, and compute their probabilities up to the current step while doing so. If a path reaches $\gamma \in F$, add its probability to α_-; if a path reaches $\gamma \in \widetilde{F}$, subtract its probability from α_+.

6.1 Necessity of Decisiveness

The under and over-approximations are clearly sound, but do they converge? Observe that we keep track of paths until they reach either F or $\widetilde{F}$, and account for their probabilities in the approximations only after they do. It is indeed possible that the approximations do not converge because of infinite runs that are "indecisive" with respect to F: they visit neither F nor $\widetilde{F}$.

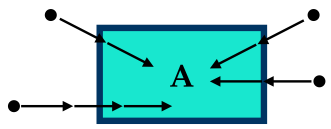
It is precisely this indecision that we seek to tackle with fairness. A Markov Chain is *decisive* [11, 6] with respect to a set of configurations F if for any configuration γ, a run starting from it is indecisive with respect to F with probability 0. Formally, for each configuration γ, it is the case that $Prob_\gamma \left(\Diamond F \vee \Diamond \widetilde{F} \right) = 1$. Put differently, if F is always reachable along a run ρ then ρ will almost certainly eventually reach F, i.e., $Prob_\gamma(\Diamond F \mid \Box \exists \Diamond F) = 1$.

Decisiveness ensures that the approximations converge and the algorithm terminates because, by definition, the problematic indecisive paths collectively have probability 0. We now turn to the enforcement of this property through fairness. In fact, we will enforce a stronger condition: that the Markov Chain itself be decisive, i.e. the property of decisiveness holds with respect to *any* set F of configurations.

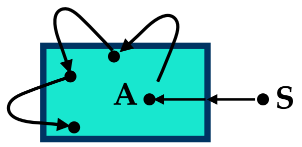
6.2 Attractors

We enforce decisiveness by asserting the existence of a *finite attractor* [10]. An *attractor* A is a set of configurations, such that each run of the system will almost certainly eventually reach A. Figure 6.2(a) illustrates an attractor. Formally, for each configuration γ of the system, we have $Prob_\gamma(\Diamond A) = 1$, i.e., we reach the set A from γ with probability one.

Attractors satisfy an even stronger condition, namely any run of the system will almost certainly visit A, not only once, but *infinitely often.* The reason (illustrated in Figure 6.2(b)) is the following. Let us consider a run ρ of the system.



(a) A: An attractor. Under probabilistic TSO, the set of configurations with empty buffers is an attractor.



(b) Repeated reachability of attractors.

Fig. 8. Attractors.

By definition of an attractor, ρ will almost certainly eventually reach a configuration $\gamma_1 \in A$. We apply the definition of an attractor to the continuation of ρ from γ_1. This continuation will almost certainly eventually reach a configuration $\gamma_2 \in A$. The reasoning can be repeated infinitely thus obtaining an infinite sequence $\gamma_1, \gamma_2, \ldots$ of configurations inside A that will be visited. This means that A will be visited infinitely often with probability 1.

By an identical line of reasoning, we can conclude that the property of decisiveness implies that $\mathit{Prob}_\gamma(\Box\Diamond F \mid \Box\exists\Diamond F) = 1$, i.e. if F is always reachable along a path, then the path almost certainly visits F not just once, but infinitely often.

Recall the description of Probabilistic Memory Fairness in Section 5.3. We declared that the set of plain configurations (for TSO, those with empty buffers) be visited infinitely often with probability 1. This precisely asserts that the set of plain configurations is an *attractor*. Moreover, when we assume that the processes have finitely many control states and that shared variables can take finitely many values, we get that the set of plain configurations is a *finite* attractor.

6.3 Sufficiency of Finite Attractors

Let us now motivate why our declaration of the existence of finite attractor suffices to enforce decisiveness. (Fig. 9). We partition A into two sets: $A_0 := A \cap \widetilde{F}$ and $A_1 := A \cap \neg\widetilde{F}$. In other words, the configurations in A_0 cannot reach F (in the underlying transition system), while from each configuration in A_1 there is a path to F. Consider a run ρ. We need to show that ρ will almost certainly eventually either reach $\widetilde{F}$ or reach F. We know that ρ will almost certainly visit A infinitely often. By construction, exactly one of A_0 or A_1 is visited infinitely often. If A_0 is visited infinitely often, we are done, because $A_0 \subseteq \widetilde{F}$. Otherwise, ρ will visit A_1 infinitely often with probability 1. Since A_1 is a finite set, with probability 1, there is a particular configuration $\gamma \in A_1$ that will be visited infinitely often by ρ. By definition, we know that F is reachable from γ, i.e., there is a path (say of length k) from γ to F. Let β be the probability that this path is taken during the next k steps of the run. This means that each time ρ

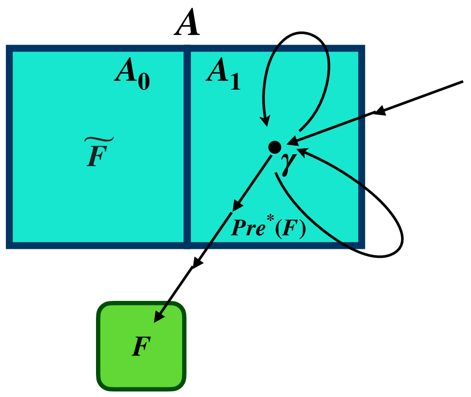


Fig. 9. Attractors and decisiveness.

visits γ, it will reach F during the next k steps with probability at least β, which implies that ρ cannot avoid F forever. Thus ρ will almost certainly eventually reach F.

The above observations allow us to use the framework of Section 5.4 and the breadth-first exploration procedure described at the beginning of this section to approximate $Prob_{\gamma_0}[\Box\Diamond F] = 1 - Prob_{\gamma_0}[\Diamond \widetilde{F}] = 1 - Prob_{\gamma_0}[\Diamond A_0]$ too. The first equality follows from decisiveness, the second from the attractor property. Here, we decrement the over-approximation when a path reaches A_0, and increment the under-approximation when a path reaches a configuration in A_1 from which A_0 is unreachable.

Notice how these queries crucially rely on the finite graph we described in Section 5.4, as well as on the probabilistic fairness properties discussed in this section for correctness.

6.4 Getting Desired Finite Attractors

In this section, we outlined a simple procedure to verify or quantify the satisfaction of liveness specifications. We identified the notion of decisiveness, a property of Markov Chains essential to the algorithm, and defined the concept of a finite attractor as a means of ensuring decisiveness. This coincided with the fairness notions we obtained from the modeling perspective. To reconcile the declarations of the modeling with the requirements of the algorithm, we now briefly discuss the assignment of transition probabilities to get the finite attractor we desire.

Recall that we consider systems that consist of finite sets of finite-state processes, communicating through shared variables according to the TSO semantics. Note that TSO uses unbounded store buffers to model message propagation: the underlying transition system is therefore infinite-state. To induce a Markov Chain, choices between different enabled transitions are resolved probabilistically, i.e. the scheduler uses a stochastic policy to pick an enabled process

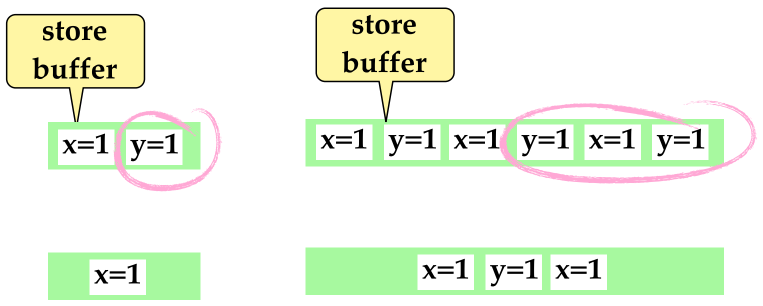


Fig. 10. Probabilistic Fairness under TSO semantics: half the buffers are flushed in expectation

to execute. Furthermore, after each process step, the program probabilistically performs memory updates as follows: from each process' store buffer, it picks a prefix whose length is chosen uniformly at random. Messages from the chosen prefixes are randomly interleaved to generate the order in which they are flushed to the main memory.

In this model, at every step, the length of each buffer is halved in expectation. Intuitively, the run almost certainly "gravitates" toward configurations with small buffers. The limit of this tendency, of course, is to (repeatedly) visit the set of configurations with *empty* buffers. In formal terms, the transition probabilities are chosen in such a way that a set of configurations with empty buffers is an *attractor*. There are only finitely many such configurations: consequently, the induced Markov chain is decisive.

7 Conclusions and Future Work

We have presented a framework for verifying liveness properties for concurrent programs running under weak memory models. To that end, we have defined a new notion of fairness corresponding to a natural restriction on program behavior and equivalent to probabilistic fairness. In particular, the latter allows reducing checking liveness properties to checking safety properties for programs running under weak memory models.

There are several directions for future work:

– Applying automata inclusion techniques, such as the one in [7], to check liveness properties efficiently.
– Designing abstraction algorithms for verifying parameterized systems, i.e., systems consisting of arbitrary many processes [8, 9, 1].
– Developing partial-order reduction technique for more scalable verification [12, 5, 2].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Parosh Aziz Abdulla. Regular model checking. *STTT*, 14(2):109–118, 2012.
2. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017.
3. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Raj Aryan Agarwal, Adwait Godbole, and S. Krishna. Probabilistic total store ordering. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 317–345. Springer, 2022.
4. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Adwait Godbole, Shankaranarayanan Krishna, and Mihir Vahanwala. Overcoming memory weakness with unified fairness - systematic verification of liveness in weak memory models. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I*, volume 13964 of *Lecture Notes in Computer Science*, pages 184–205. Springer, 2023.
5. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. Optimal stateless model checking for reads-from equivalence under sequential consistency. *PACMPL*, 3(OOPSLA), 2019.
6. Parosh Aziz Abdulla, Nathalie Bertrand, Alexander Moshe Rabinovich, and Philippe Schnoebelen. Verification of probabilistic systems with faulty communication. *Inf. Comput.*, 202(2):141–165, 2005.
7. Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukás Holík, Chih-Duo Hong, Richard Mayr, and Tomás Vojnar. Advanced ramsey-based büchi automata inclusion testing. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2011.
8. Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. Parameterized verification through view abstraction. *STTT*, 18(5):495–516, 2016.
9. Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Handling parameterized systems with non-atomic global conditions. In Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, volume 4905 of *Lecture Notes in Computer Science*, pages 22–36. Springer, 2008.
10. Parosh Aziz Abdulla, Noomene Ben Henda, and Richard Mayr. Verifying infinite markov chains with a finite attractor or the global coarseness property. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 127–136. IEEE Computer Society, 2005.
11. Parosh Aziz Abdulla, Noomene Ben Henda, and Richard Mayr. Decisive markov chains. *LMCS*, 3(4), 2007.

12. Parosh Aziz Abdulla, Bengt Jonsson, Mats Kindahl, and Doron Peled. A general approach to partial order reductions in symbolic verification. In *CAV 98*, volume 1427 of *LNCS*, pages 379–390, 1998.
13. Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distrib. Comput.*, 9(1):3749, mar 1995.
14. Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 718, New York, NY, USA, 2010. Association for Computing Machinery.
15. Manfred Broy and Martin Wirsing. On the algebraic specification of nondeterministic programming languages. In *CAAP*, 1981.
16. Edward Y. Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In Werner Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 474–486. Springer, 1992.
17. Rob Glabbeek and Peter Hfner. Progress, justness, and fairness. *ACM Computing Surveys*, 52:1–38, 08 2019.
18. Ori Lahav and Udi Boker. Whats decidable about causally consistent shared memory? *ACM Trans. Program. Lang. Syst.*, 44(2), apr 2022.
19. L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28:690–691, 1979.
20. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification.* Springer, 1992.
21. CORPORATE SPARC International, Inc. *The SPARC Architecture Manual (Version 9).* Prentice-Hall, Inc., USA, 1994.
22. Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL*, pages 184–193. ACM Press, 1986.