Verification under causally consistent shared memory

Ori Lahav, Tel Aviv University

We consider concurrent programs interacting with causally consistent shared memory. After describing the semantics of such programs, we outline several verification problems and survey some existing solutions.

1. INTRODUCTION

The canonical, often implicit, model of shared-memory concurrency is *sequential consistency* (SC) [Lamport 1979]: the behaviors of a concurrent program are assumed to be the result of some interleaving of the operations of the different threads. While this model is intuitive and well-known, no mainstream multiprocessor actually provides sequential consistency. Instead, to have better performance, modern chips (e.g., Intel x86, IBM POWER, and ARM) employ different optimizations such as local write buffers, hierarchies of caches, and speculative execution. Consequently, without explicit costly synchronization, they exhibit "weak" (non-SC) behaviors, which are not a result of any interleaving of the operations of the different processors. Accordingly, the specifications of programming languages, like C and C++, formulate *weak memory models* that should be assumed by their clients, allowing them to demand SC when they need it, but also support a range of cheaper memory operations.

This raises interesting challenges and opportunities for formal verification: How should one verify the correctness of a program running under a weak memory model? How does one adapt the reasoning principles and verification methods that were developed for SC?

In this paper, we focus on shared memory programs running under *causal consistency*. Most of the paper is devoted to the introduction of this weak memory model. In the second part, we describe several verification problems and survey some results and open problems in this field.

Causal consistency constitutes a natural weakening of SC. Roughly speaking, while SC requires one global order of all operations, causal consistency only requires causally dependent operations to appear in one global order, thus, allowing threads to disagree on the order of causally independent operations. Causal consistency originally arose in distributed systems [Ahamad et al. 1995], where the nodes communicate by message passing and avoid costly global synchronization. Nevertheless, nowadays, causal consistency plays a prominent role in shared memory systems. For example, certain forms of causal consistency are provided by the release/acquire fragment of the C/C++11 memory model [ISO/IEC 9899:2011 2011; ISO/IEC 14882:2011 2011; Batty et al. 2011] (when all accesses to shared variables are annotated with release and acquire access modes) and by the POWER multi-processor [Alglave et al. 2014] (when its so-called "lightweight" and "instruction" fences are placed before every shared store and after every shared load, respectively). The x86-TSO model [Owens et al. 2009] provides causal consistency "for free". That is, even without additional barriers, x86-TSO provides stronger guarantees than causal consistency.

It should be also noted that while causal consistency allows higher performance implementations, its guarantees are often sufficiently strong for the correctness of concurrent algorithms. In particular, it supports the common "message passing" idiom, which appears in various synchronization mechanisms (see MP below).

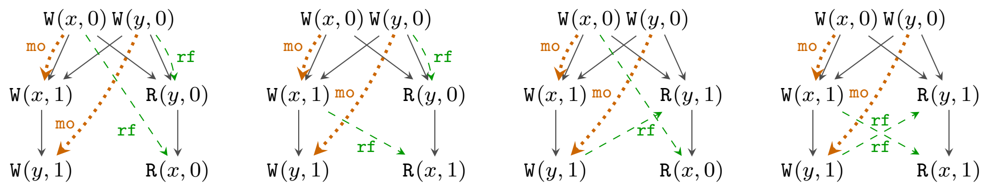
2. WHAT IS A CAUSALLY CONSISTENT SHARED MEMORY?

In this section, we define causally consistent shared memory. In fact, several natural variants have been studied. While traditional models are given by an operational semantics with constructs like buffers, messages and timestamps, it is easier to define the different causal consistency models *declaratively*, abstracting away the implementation details. In this approach, the program is first associated with a set of execution graphs, each of which summarizes a particular program run and describes all accesses to shared variables and the relations between them in that run. Then, formal constraints are used to filter out *inconsistent* execution graphs, and the remaining *consistent* graphs are defining all possible program behaviors. Different variants of causal consistency are obtained by imposing different consistency constraints.

As a first example, consider the following program:

$$\begin{array}{l} x := 1 \\ y := 1 \end{array} \; \Bigg\| \; \begin{array}{l} a := y \\ b := x \end{array} \qquad \text{(MP)}$$

Here and henceforth we assume that all variables are initialized to 0, and use $x, y, \ldots$ for shared variables (also called *locations*), and $a, b, \ldots$ for local variables. The following are four possible execution graphs of this program:



(i) (ii) (iii) (iv)

The nodes of the execution graphs, called *events*, represent accesses to the shared memory (including the implicit initialization writes). In our formalism, which closely follows the C/C++11 model, three binary relations relate the events of the graph: (a) the *program order* relation ($\mathtt{po}$), depicted by solid edges, tracks the order in each thread (with initialization events preceding all other events); (b) the *reads-from* relation ($\mathtt{rf}$), depicted by dashed edges, associates *every* read event with the write event it reads from; and (c) the *modification order* relation ($\mathtt{mo}$), depicted by dotted edges, totally orders the writes to each location. The modification order $\mathtt{mo}$ is used to globally decide on the order of concurrent writes to the same location (in this simple example, there are no concurrent writes). We do not include here the formal definition of the set of graphs associated with a given program, but the intention should be clear.

Before turning to the consistency constraints of causal consistency, it is instructive to see a simple definition of SC in this declarative framework. SC can be straightforwardly defined by requiring the existence of a total order T such that (i) T extends $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo}$; and (ii) every read r reads from the T-last write to the same location that is T-before r (that is, if $\langle w, r\rangle \in \mathtt{rf}$ then there does not exist w' writing to the same location as w such that $\langle w, w'\rangle \in T \wedge \langle w', r\rangle \in T$). It is easy to check that execution graphs (i), (ii) and (iv) of the MP program above are SC-consistent, while execution graph (iii) (yielding $a = 1$ and $b = 0$) is not SC-consistent (as the total order forces $\mathtt{W}(x, 1)$ to be the the last write to x before $\mathtt{R}(x, 0)$).

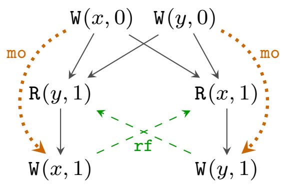
REMARK 1. Such a total order T exists iff $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup (\mathtt{rf}^{-1} ; \mathtt{mo})$ is *acyclic* (where ';' denotes relation composition and $\mathtt{rf}^{-1}$ denotes the inverse of $\mathtt{rf}$). This provides a more "efficient" declarative formulation of SC that refrains from existentially quantifying over a total order T, and uses instead the already existing order on writes $\mathtt{mo}$.

Casual consistency has much weaker requirements. First, in all causal consistency models, one requires that

$$\mathtt{po} \cup \mathtt{rf} \text{ is acyclic,}$$

and therefore, $(\texttt{po} \cup \texttt{rf})^+$ (that is, the transitive closure of $\texttt{po} \cup \texttt{rf}$) is a (strict) partial order. In causal consistency models, this order is often called *happens-before* and denoted by $\texttt{hb}$. The acyclicity condition forbids so-called "load-buffering" behaviors, which are allowed in weaker models:

$$\begin{array}{l||l} a := y \quad /\!/1 & b := x \quad /\!/1 \\ x := 1 & y := 1 \end{array}$$



(LB)

The outcome annotated in the program comments ($a = b = 1$) can be only explained by a $\texttt{po} \cup \texttt{rf}$-cycle (shown in the depicted execution graph), and is disallowed by causal consistency.

The other consistency constraints in causal consistency are needed to ensure that threads never read from a certain write event when they are aware of a later write event to the same location. There is more than one way to precisely interpret this requirement, and thus there are several variants of causal consistent memory. We present three natural models (in increasing "strength") and relate them to different models previously introduced in the literature. We note that the difference between these models only appears in execution graphs with *write-write races*, namely executions containing two write events w_1, w_2 that write to the same location and are $\texttt{hb}$-incomparable ($\langle w_1, w_2 \rangle \notin \texttt{hb}$ and $\langle w_2, w_1 \rangle \notin \texttt{hb}$). If no execution graph of a given program has such race (as in MP above), then the three models presented next coincide.

2.1. The three models

2.1.1. Weak release acquire (WRA). The weakest model we consider, denoted WRA (for *weak release acquire*), can be shown to be equivalent to the basic causal consistency model (called CC) in [Bouajjani et al. 2017]. It requires that read events never read from a write event when they are aware of a later write event to the same location, where "aware" and "later" are interpreted using $\texttt{hb}$. Formally, for every read r from location x and two writes w, w' to x, we should never have $\langle w, r \rangle \in \texttt{rf} \wedge \langle w, w' \rangle \in \texttt{hb} \wedge \langle w', r \rangle \in \texttt{hb}$. This condition excludes execution graph (iii) of the MP program above: the read from x is aware (in $\texttt{hb}$) of a write to x that is ($\texttt{hb}$-) later than the one it reads from.

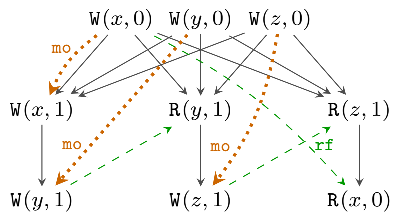
A convenient and concise way to express this condition is to require that

$$[\texttt{W}] \,;\, \texttt{hb}|_{\texttt{loc}} \,;\, [\texttt{W}] \,;\, \texttt{hb} \,;\, \texttt{rf}^{-1} \text{ is irreflexive.}$$

In this expression, in addition to the notations from Remark 1, $\texttt{hb}|_{\texttt{loc}}$ denotes the restriction of $\texttt{hb}$ to accesses of the same location, and $[\texttt{W}]$ denotes the identity relation on all write events.

Roughly speaking, the transitivity of $\texttt{hb}$ ensures that when thread τ reads from some write event w of thread π, thread τ becomes "aware" of whatever thread π was aware of at the time of writing w. In particular, in the following example, the annotated outcome is disallowed under WRA (the depicted execution graph is WRA-inconsistent):

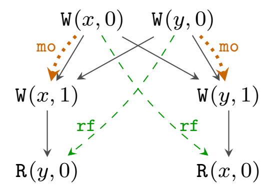
$$\begin{array}{l||l||l} x := 1 & a := y \quad /\!/1 & b := z \quad /\!/1 \\ y := 1 & z := 1 & c := x \quad /\!/0 \end{array}$$



(MP-trans)

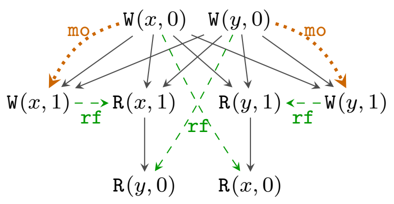
WRA allows every outcome allowed by SC. Indeed, the total order T required by SC has to satisfy both that $\texttt{hb} \subseteq T$ and that $[\texttt{W}] ; T|_{\texttt{loc}} ; [\texttt{W}] ; T ; \texttt{rf}^{-1}$ is irreflexive. The existence of such a relation T clearly implies the consistency condition of WRA. (Similar arguments show that the other models below allow every outcome allowed by SC.) The following standard litmus tests demonstrate that WRA is *strictly* weaker than SC:

$$\begin{array}{l||l} x := 1 & y := 1 \\ a := y \quad /\!/0 & b := x \quad /\!/0 \end{array}$$



(SB)

$$\begin{array}{l||l||l||l} x := 1 & a := x \quad /\!/1 & c := y \quad /\!/1 & y := 1 \\ & b := y \quad /\!/0 & d := x \quad /\!/0 & \end{array}$$



(IRIW)

Both execution graphs are WRA-consistent, and thus the annotated outcomes are allowed under WRA. On the other hand, they are clearly disallowed by SC. While they may seem counterintuitive when thinking in terms of SC, these outcomes are actually quite natural when thinking about a (truly) concurrent system. For example, in SB, the outcome $a = b = 0$ is unavoidable if we allow the threads to run without enforcing any communication between them. In addition, the IRIW example shows that WRA is *non-multi-copy-atomic*: different threads may observe different writes in different orders. (This is a crucial difference between causal consistency models and the x86-TSO model [Owens et al. 2009], which is multi-copy-atomic and forbids IRIW's outcome.) In these two examples, as in MP and MP-trans above, there are no write-write races, and, thus, the stronger models defined below allow these outcomes as well.

Note that the $\texttt{mo}$ relation does not play any role in WRA. In fact, for WRA, we can simply exclude $\texttt{mo}$ from execution graphs. Below, we refer to execution graphs with only $\texttt{po}$ and $\texttt{rf}$ as *reduced*.

2.1.2. Release acquire (RA). When there are write-write races, WRA places almost no guarantees. For example, in the following programs, WRA allows the annotated outcomes:

$$\begin{array}{l||l||l} x := 1 & x := 2 & a := x \quad /\!/1 \\ & & b := x \quad /\!/2 \\ & & c := x \quad /\!/1 \end{array} \qquad\qquad \begin{array}{l||l} x := 1 & x := 2 \\ a := x \quad /\!/2 & a := x \quad /\!/1 \end{array} \qquad \text{(WWrace)}$$

These outcomes are *not* observable on hardware such as POWER and ARM, and the C/C++11 release/acquire fragment, denoted here by RA, forbids this outcome as well. The RA model utilizes the $\texttt{mo}$ relation in execution graphs to (globally) decide on which write is later. By definition, $\texttt{mo}$ is required to be a union of total orders, each of which orders all writes to a given location. Using $\texttt{mo}$, RA is formulated by two consistency constraints:

$$\texttt{mo} ; \texttt{hb} \text{ is irreflexive} \qquad\qquad \text{and} \qquad\qquad \texttt{mo} ; \texttt{hb} ; \texttt{rf}^{-1} \text{ is irreflexive.}$$

The first condition requires that whenever $\texttt{hb}$ orders two writes to the same location, then $\texttt{mo}$ has to agree with $\texttt{hb}$ on the order of these writes. The second condition is similar to the condition of WRA, replacing $[\mathtt{W}] ; \mathtt{hb}|_{\mathtt{loc}} ; [\mathtt{W}]$ with $\mathtt{mo}$, thus requiring that read events never read from a write when they are aware (in $\mathtt{hb}$) of an $\mathtt{mo}$-*later* write.

Note that the first condition implies that $[\mathtt{W}] ; \mathtt{hb}|_{\mathtt{loc}} ; [\mathtt{W}] \subseteq \mathtt{mo}$, and so RA is at least as strong as WRA. It is *strictly* stronger, as, for instance, the outcomes annotated in the WWrace programs above are disallowed by RA. To see this, observe that for both options for $\mathtt{mo}$ (either from $\mathtt{W}(x, 1)$ to $\mathtt{W}(x, 2)$ or vice-versa), we have $\langle w, w\rangle \in \mathtt{mo} ; \mathtt{hb} ; \mathtt{rf}^{-1}$ where w is the $\mathtt{mo}$-earlier write.

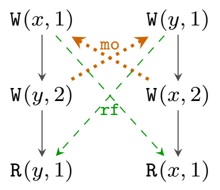
REMARK 2. As shown in [Lahav and Vafeiadis 2015, Appendix B], it is possible to define RA using *reduced* execution graphs (that do not include the $\mathtt{mo}$ relation). That is, one can identify conditions on $\mathtt{po}$ and $\mathtt{rf}$ that are necessary and sufficient for the existence of an $\mathtt{mo}$ relation that meets the requirements of RA. Concretely, let $\mathtt{wb}$ (standing for *writes before*) be the relation defined by

$$\mathtt{wb} \triangleq [\mathtt{W}] ; \mathtt{hb}|_{\mathtt{loc}} ; (\mathtt{rf}^{-1})^{?} ; [\mathtt{W}] \ \setminus \ [\mathtt{W}],$$

where $(\mathtt{rf}^{-1})^{?}$ denotes the reflexive closure of $\mathtt{rf}^{-1}$. It is not hard to show that $\mathtt{wb}$ is acyclic in every RA-consistent execution (by showing that $\mathtt{wb} \subseteq \mathtt{mo}$), and that any total order extending $\mathtt{wb}$ may serve as the $\mathtt{mo}$ relation in an RA-consistent execution. Thus, we could equivalently define the RA-allowed outcomes of a program based on the set of *reduced* execution graphs of the program in which $\mathtt{wb}$ is acyclic.

2.1.3. Strong release acquire (SRA). The following annotated outcome is allowed by RA (for brevity, the initialization writes are omitted from the execution graph):

$$\begin{array}{l||l} x := 1 & y := 1 \\ y := 2 & x := 2 \\ a := y \ \ /\!\!/ 1 & b := x \ \ /\!\!/ 1 \end{array} \qquad \text{(2+2W)}$$



Observing that hardware does not exhibit this outcome, Lahav et al. [2016] proposed to strengthen C/C++11's RA model with the following condition:

$$\mathtt{hb} \cup \mathtt{mo} \text{ is acyclic.}$$

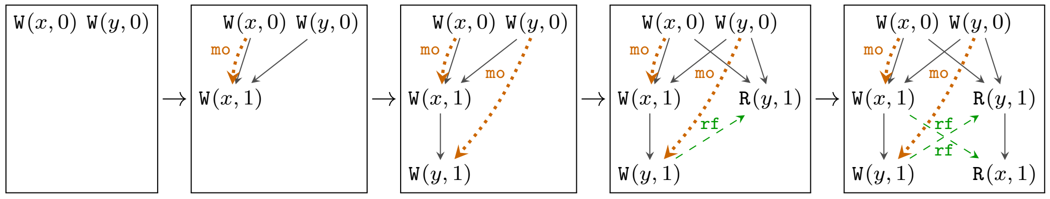
To obtain the annotated outcome of the 2+2W example ($a = b = 1$), the $\mathtt{mo}$ edges are forced to create a $\mathtt{po}\cup\mathtt{mo} \subseteq \mathtt{hb}\cup\mathtt{mo}$ cycle (or else we will invalidate RA's constraints above). Thus, the additional condition forbids the annotated outcome. Clearly, it is stronger than RA's condition that only requires that $\mathtt{mo} ; \mathtt{hb}$ is irreflexive. The strengthened model, referred to as SRA (for strong release acquire), can be shown to be equivalent to the causal convergence (CCv) model in [Bouajjani et al. 2017] and to the causal consistency models presented in [Burckhardt 2014; Cerone et al. 2015]. (These models also handle general transactions including more than one instruction, which are not considered in this paper.)

Lahav et al. [2016] proved that, for the declarative POWER model presented in [Alglave et al. 2014], and the existing compilation scheme of C/C++11 to POWER [Mapping 2016], SRA is the "best one can have". That is, not only that every outcome of the compiled program that is allowed by POWER model is allowed for the source program by SRA, but we also have the converse: every outcome allowed for the source program by SRA is allowed by POWER for the compiled program. Hence, it is not possible to further strengthen the semantics of release/acquire in C/C++11 without modifying the compilation scheme to POWER and paying the induced performance cost.

2.2. Operational semantics

We presented the different causal consistency models using a *declarative* (also called *axiomatic*) semantics. Nevertheless, for the models defined above, the declarative presentation can be easily "operationalized", leading to an equivalent interleaving-based operational semantics. A simple way to do so is to take the states of the operational semantics to consist of a program, local stores for each thread (assigning values to local variables), and a consistent execution graph. The initial state consists of the input program, the initial store for each thread, and the initial execution graph (which includes only the initialization writes). Transitions reduce one thread at a time, appropriately updating the state. In particular, steps that involve accesses to the global memory extend the current execution graph with one event that is placed po-after all other events of the thread taking the step. A transition is only possible if the current rf and mo relations can be also extended in a way that maintains consistency.

For example, for any of the models above, the $a = b = 1$ outcome of the MP program can be obtained by the following run (we only depict the execution-graph component of the state):



It is easy to see that such operational semantics is equivalent to the declarative semantics it is derived from. First, every execution graph that is reachable by the operational semantics is consistent, and so the outcomes according to the operational semantics are all allowed by the declarative one. Second, the different causal consistency models are *closed under* $\mathtt{po} \cup \mathtt{rf}$*-prefixes*, that is: the restriction of a consistent execution graph to a set of events that is downwards-closed with respect to $\mathtt{po} \cup \mathtt{rf}$ is always consistent. Thus, every outcome that is allowed by the declarative semantics can be obtained (typically in more than one way) by its operationalized counterpart.

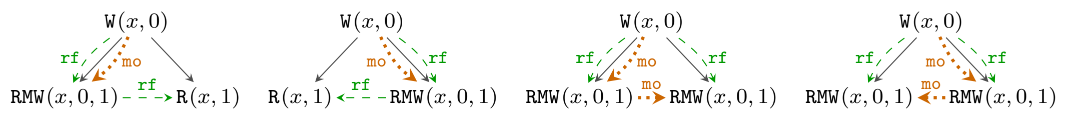
Two possible simplifications of the operationalized declarative semantics are possible. First, one can identify the exact local conditions on execution graphs that preserve consistency of each step instead of directly referring to the consistency condition. For example, under RA, a step of thread τ reading v from x should add a read event r po-after all events of thread τ and extend rf with one pair $\langle w, r\rangle$, where w is some write event in the current graph writing v to x and satisfying $\langle w, e\rangle \notin \mathtt{mo} \mathbin{;} \mathtt{hb}^?$ for every event e of thread τ ($\mathtt{hb}^?$ denotes the reflexive closure of hb). Second, it is possible to extract from the execution graph the information that is actually required to decide whether the possible steps are allowed or not, thus using (and maintaining) a more concise representation of the state. For example, in RA, instead of keeping the whole execution graph, it suffices to maintain the set of all write events in the execution graph, together with the mo relation between them, as well as the last write to every location that was observed by each thread. Following this approach, Kang et al. [2017] and Kaiser et al. [2017] describe a semantics for RA where write events correspond to timestamped messages, and the order between the timestamps represents the mo relation. In turn, instead of execution graphs, states include a message pool, containing one message for every write that was performed, as well as mappings (called *thread views*) that assign to each thread τ and location x, the last timestamp that τ observed for x.

2.3. Read-modify-write instructions

The models presented above include reads and writes, leaving out *read-modify-write* (RMW) instructions. These instructions "atomically" perform a read possibly followed, depending on the value that was read, by a write. The value written (if any) may also depend on the value that was read. RMWs are indispensable in shared memory concurrent programming. In fact, for the models above, they are necessary for implementing a critical section (observing that write-read reordering is sound in these models, this follows from the results in [Attiya et al. 2011]).

To give (declarative) semantics to RMWs, one first introduces corresponding events in execution graphs, with labels of the form $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$ where x is the location being accessed, and $v_{\mathtt{R}}, v_{\mathtt{W}}$ are (respectively) the read and written values. (Equivalently, it is possible to treat RMWs as a pair of a read and write events and include a special relation marking pairs that belong to the same RMW instruction.) As read events, RMW events require an incoming rf edge, while as write events, RMW events participate in the mo order. For example, we have the following execution graphs for the following program (CAS denotes an atomic compare-and-swap instruction, returning the value of the variable that was read):

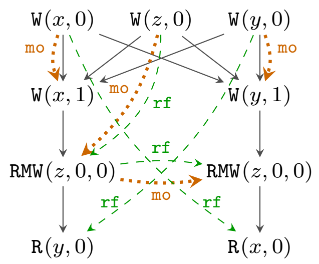
$$a := \mathtt{CAS}(x, 0, 1) \;||\; b := \mathtt{CAS}(x, 0, 1) \qquad \text{(2RMW)}$$



As before, there is more than one option for interpreting the atomicity condition that, in particular, should rule out the $a = b = 0$ outcome in the 2RMW example, where both CAS instructions succeed in swapping the value (as happens in the two right execution graphs). A weak condition may only require that different RMW events never read from the same event. This condition can be naturally added to WRA, as it does not refer to the mo relation. A stronger condition (employed in the RA and SRA models) requires that RMW events read from their immediate mo-predecessor.

Both conditions suffice for implementing a spinlock using CAS instructions. Furthermore, they allow RMWs to serve as *fences* that can be used to forbid some weak behaviors. For example, to forbid the $a = b = 0$ of the SB program above, one may place an RMW of an otherwise unused location (e.g., $\mathtt{CAS}(z, 0, 0)$ or $\mathtt{FetchAndIncrement}(z)$) between the two instructions in each thread:

$$\begin{array}{l} x := 1 \\ c := \mathtt{CAS}(z, 0, 0) \\ a := y \quad /\!/ 0 \end{array} \;\Bigg|\Bigg|\; \begin{array}{l} y := 1 \\ d := \mathtt{CAS}(z, 0, 0) \\ b := x \quad /\!/ 0 \end{array} \qquad \text{(SB+fences)}$$



In consistent executions, one of the two RMW events will have to read from the other (as they may not read from the same event), making one of the reads hb-after the write of the other thread, so, under any of the casual consistency models, it may not read the initial value. For example, in the depicted execution, we have hb from $\mathtt{W}(x, 1)$ to $\mathtt{R}(x, 0)$, while $\mathtt{R}(x, 0)$ reads from an hb-earlier (as well as mo-earlier) write event. Thus, this execution graph is inconsistent (under any of the casual consistency models). As shown by Lahav et al. [2016], for any program, placing such fences between every pair of instructions suffices to restore SC. Lahav et al. [2016] further identified a certain class of programs, which they call "server-client programs", in which SC can be restored with fewer fences.

3. VERIFICATION PROBLEMS AND EXISTING SOLUTIONS

Having defined different causal consistency semantics, we now describe several natural verification problems and briefly survey some of the existing solutions and open problems. We keep the presentation on the intuitive level and refer the reader to relevant papers for the formal definitions and theorems. While we focus on causal consistency, the same verification problems arise for any weak memory model. Most of the works mentioned below target more general models that allow several access modes and include causal consistency as a principal fragment.

3.1. Unbounded verification

When every thread is a finite-state program (i.e., it has a finite data domain but may include loops), then verification of a concurrent program under SC is clearly decidable. Here, we refer to traditional verification of safety properties, formulated by local assertions on the values of the local variables at different program points. This verification problem amounts to state reachability problem in a labeled transition system capturing all interleaved runs of the concurrent program. Assuming SC semantics, this problem is PSPACE-complete [Kozen 1977].

Now, if instead of SC we take any of the causal memory models described above, the decidability of this problem is not at all clear. Indeed, even when the data domain is bounded, the states of the system (see §2.2 above) record the whole execution history, and, for programs with loops, their number is unbounded. Very recently, Abdulla et al. [2019] showed that this reachability problem is undecidable for RA (including RMWs) by reduction from Post's correspondence problem. The same reachability problem was shown to be decidable (with non-primitive recursive complexity) for the x86-TSO memory model, and undecidable for certain models based on instruction reorderings [Atig et al. 2012, 2010; Abdulla et al. 2018a]. It is unclear whether the techniques of these papers can be used for the other causal models as well, and, to the best of our knowledge, the decidability of reachability under WRA and SRA is still open.

3.2. Bounded model checking

A common approach to avoiding the complexity of unbounded verification is to restrict ourselves to bounded runs, by first unfolding the loops up to a certain bound. Now, given a loop-free program, verifying assertion violations is clearly decidable, as the number of possible execution graphs is bounded. Naively generating all possible consistent execution graphs is however inefficient, in terms of time and memory.

Recently, several works developed more efficient techniques for this problem under weak memory semantics. In particular, Kokologiannakis et al. [2017]; Abdulla et al. [2018b] developed algorithms and tools for (bounded) model checking under RA. (Kokologiannakis et al. [2017] also addressed the WRA model, as well as the full RC11 model—a repaired and strengthened version of the C/C++11 model [Lahav et al. 2017].) Inspired by stateless partial order reduction techniques (for SC), these algorithms efficiently generate all consistent execution graphs of a given program without storing all of them in memory. Loosely speaking, based on the operationalized RA semantics, one begins by generating some RA-consistent execution graph, while tracking the arbitrary choices that were made, and later backtracks and reverts these choices.

Notably, these methods perform better for RA than similar methods for SC. To obtain the better performance, two main properties that distinguish RA from SC were identified and utilized:

— The first property, defined and used in [Kokologiannakis et al. 2017], is that RA-consistency is *prefix-determined*. That is, if r is a po-maximal read event in an execution graph G, then the RA-consistency of G follows from the consistency of G without r and the consistency of the execution graph obtained by restricting G to the hb-prefix of r. This property does not hold for SC. To see this, consider the execution graph depicted in the SB example above, and let r denote $\mathtt{R}(x, 0)$ event. Without r, it is SC-consistent. In addition, the hb-prefix of r (including r itself) is SC-consistent. Nevertheless, the whole execution graph is not SC-consistent. In [Kokologiannakis et al. 2017], prefix-determinedness turns out to be useful for the efficient exploration of consistent executions. Roughly speaking, it allows one to safely revert an earlier rf-choice without revisiting the decisions made for concurrent events.

— The second property, utilized in [Abdulla et al. 2018b], concerns the complexity of deciding whether a reduced execution graph (including only po and rf, but not mo) is consistent (that is, whether there exists an mo relation that will make the reduced graph consistent). While deciding whether a given reduced execution graph is SC-consistent is NP-complete [Gibbons and Korach 1997], following Remark 2, RA-consistency can be easily decided in polynomial time. In [Abdulla et al. 2018b], this fact allows efficient exploration of RA-consistent *reduced* execution graphs. Since program assertions cannot directly observe mo, exploring reduced execution graphs suffices for verification. For programs with write-write races, their number may be significantly smaller than the number of execution graphs.

3.3. Program logics

Weak memory models pose interesting challenges to Hoare-style verification as well (see also [Vafeiadis 2017]). First, the following example from [Lahav and Vafeiadis 2015] demonstrates that, even without ghost variables, the basic Owicki-Gries logic (OG, for short) [Owicki and Gries 1976] is unsound for causal consistency models:[1]

$$
\begin{array}{l}
\quad\{x = 0 \land y = 0 \land a \neq 0\} \\
\begin{array}{l||l}
\{a \neq 0\} & \{\top\} \\
x := 1 & y := 1 \\
\{x \neq 0\} & \{y \neq 0\} \\
a := y & b := x \\
\{x \neq 0\} & \{y \neq 0 \land (a \neq 0 \lor x = b)\}
\end{array} \\
\quad\{a \neq 0 \lor b \neq 0\}
\end{array}
$$

Indeed, this proof outline is valid in OG, and yet, the outcome $a = b = 0$ is allowed by causal consistency (see the SB example above). In particular, note that the assertion $y \neq 0 \land (a \neq 0 \lor x = b)$ is stable under the preconditioned assignment $\{x \neq 0\} a := y$ since we have $y \neq 0 \land (a \neq 0 \lor x = b) \land x \neq 0 \vdash y \neq 0 \land (y \neq 0 \lor x = b)$.

Intuitively speaking, under a weak memory model different threads may have different views of the memory, and it is generally unsound to conjoin their assertions. Furthermore, without a global state in the form of a mapping from locations to values, even the meaning of an assertion like $y \neq 0$ (where y is a shared variable) is a priori unclear. Lahav and Vafeiadis [2015] interpret such assertion placed at a specific program point in thread τ by requiring that if thread τ were to read y at this program point

[1] With ghost variables, OG is a complete proof system for SC, and, thus, it is clearly unsound for weaker models.

it would not be able to get 0. Using this interpretation, Lahav and Vafeiadis [2015] developed OGRA, a certain weakening of OG, and proved it to be sound under the RA model. Interestingly, OGRA, which requires a stronger non-interference condition than OG and restricts the use of ghost variables, still allows all OG proofs in which threads "mind their own business", that is, the proof of each thread does not mention local (or ghost) variables of other threads.

An alternative deductive approach for verification under RA (although not formulated as a Hoare logic) was developed in [Doherty et al. 2019]. The idea there is to extend the assertion language with more predicates carrying information about the execution-graph component of the state (see §2.2). In particular, Doherty et al. [2019] add "variable-ordering predicates", which have no direct analogue in deductive verification under SC, and show how these can be used to reason about programs under RA.

The method of [Alglave and Cousot 2017], which is parametric in the declarative consistency constraints, goes further in encoding the execution-graph structure in logical invariants. Using so-called "pythia variables" to give unique names to the values of read events, its invariants can precisely express the reads-from relation. This allows the proof method to be (relatively) complete.

Concurrent separation logic (CSL) [O'Hearn 2007] was also extended and adapted for weak memory models, again focusing on the RA model in particular. Compared to OG, CSL-style reasoning is closer in spirit to the causal consistency guarantees. First, CSL generally targets data-race-free programs, where SC and the causal models coincide (see also [Ferreira et al. 2010]). Second, in the presence of data races, CSL reasoning is based on "ownership transfer"—transferring from one thread to another the right to access certain locations or invariants on the state. Roughly speaking, since the causal consistency models guarantee the correctness of the message passing idioms (see examples MP and MP-trans above), ownership transfer constitutes a sound reasoning principle for causal consistency.

Vafeiadis and Narayan [2013] introduced the first specialization of CSL for a weak memory model. Their program logic, called RSL (relaxed separation logic), targets a fragment of the C/C++11 model that contains RA. Its soundness proof provides a novel semantics for assertions and Hoare-triples that refers to a non-standard notion of a state in the form of an execution graph. Following similar ideas in its soundness proof, a more expressive logic, called GPS (standing for ghost state, protocols, and separation) was developed and used to verify several challenging algorithms [Turon et al. 2014; Tassarotti et al. 2015]. In addition, a similar logic, called iGPS, was developed in the Iris framework [Kaiser et al. 2017]. In the soundness proof of this logic, the authors introduced a timestamp-based operational semantics of the memory model, which follows the ideas outlined in §2.2.

Crucially, besides release/acquire accesses, these CSL-style logics also handle C/C++11's non-atomic accesses, typically used for "data variables" (unlike "synchronization variables"). A data race involving a non-atomic access implies an undefined behavior in C/C++11, and thus non-atomic accesses allow very efficient implementation. In RSL, GPS and iGPS, a complete proof of a program (even with a trivial specification) implies its safety, which in particular means that there are no data races on non-atomic accesses.

For the synchronization variables (on which races are unavoidable), these program logics target RA, and it is interesting to see how they can be weakened for WRA, or, perhaps, strengthened for SRA. In the case of RSL, following its (mechanized) soundness proof, it is easy to see that *without any weakening* RSL is actually sound for the WRA model. In [Kokologiannakis et al. 2017], it was conjectured that GPS and its iGPS variant are also sound for WRA. OGRA, however, is able to reason about concurrent writes and will have to be weakened to obtain soundness for WRA.

3.4. Robustness verification

Another natural way to verify the correctness of a given program under weak memory semantics is to use existing techniques to verify the program assuming SC semantics and prove that the program does not have weak behaviors (behaviors that are not allowed by SC). The latter property is often called *robustness* against a certain model [Bouajjani et al. 2011]. While this approach is necessarily partial (not all weak behaviors are bugs), robustness against the causal consistency models can be often established for existing algorithms, which allows one to port algorithms that were designed for SC to a causally consistent memory. In addition, non-robust programs can be made robust by placing fences or by strengthening certain reads and writes to be RMW operations (see §2.3).

A natural problem is thus the verification of robustness against the causal consistency models. To precisely state this problem, one first has to define what constitutes a behavior of a concurrent program. If we identify the possible behaviors with the set of reachable program states (where states ascribe values to all local variables and the program counter of each thread), then, following [Derevenetc 2015, Thm. 2.12], it is easy to show that verifying robustness is as hard as the general state reachability problem (described in §3.1). We refer to this robustness definition as *state robustness*. More precise notions of a behavior induce stronger notions of robustness, which imply state robustness. In particular, one may identify program behaviors with consistent execution graphs. The induced robustness notion against a declarative model X, called *execution-graph robustness against* X, means that all X-consistnet execution graphs of the program are also SC-consistent.

Recently, Lahav and Margalit [2019] established the decidability of execution-graph robustness against the RA model (for programs with bounded data domain) and further showed that this problem is PSPACE-complete. (Execution-graph robustness against x86-TSO is of the same complexity [Bouajjani et al. 2013]). The main idea there (as well as in initial works on robustness against x86-TSO [Burckhardt and Musuvathi 2008]) is to reduce robustness verification to a state reachability problem under a (finite state) instrumented SC memory. The instrumented memory keeps track of certain properties of the generated execution graph that are used for monitoring that all steps preserving RA-consistency are also allowed by SC. It is interesting to see whether such approach can handle other models (in particular, WRA and SRA), and how can execution-graph robustness be weakened to become closer to state robustness while still maintaining a PSPACE solution.

Finally, robustness against causal consistency was also studied in the context of distributed systems, where programs typically include transactions containing more than one memory instruction (see, e.g., [Bernardi and Gotsman 2016; Nagar and Jagannathan 2018; Brutschy et al. 2018]). In this context, SC is replaced by *serializability*, which requires the atomicity of each transaction. These works provide practical over-approximations of robustness, rather than precise verification methods.

ACKNOWLEDGMENTS

I would like to thank Udi Boker, Nachum Dershowitz, and Viktor Vafeiadis for useful comments on an earlier version of this paper. The author is supported by the Israel Science Foundation (grant number 5166651), by the Alon Young Faculty Fellowship and by Len Blavatnik and the Blavatnik Family foundation.

REFERENCES

Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *PLDI (accepted for publication)*.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2018a. A load-buffer semantics for total store ordering. *Logical Methods in Computer Science* Volume 14, Issue 1 (Jan. 2018). DOI:http://dx.doi.org/10.23638/LMCS-14(1:9)2018

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018b. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. DOI:http://dx.doi.org/10.1145/3276505

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995), 37–49.

Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models. In *POPL*. ACM, New York, 3–18. DOI:http://dx.doi.org/10.1145/3009837.3009883

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. DOI:http://dx.doi.org/10.1145/2627752

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *POPL*. ACM, New York, 7–18. DOI:http://dx.doi.org/10.1145/1706299.1706303

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2012. What's decidable about weak memory models?. In *ESOP*. Springer-Verlag, Berlin, Heidelberg, 26–46. DOI:http://dx.doi.org/10.1007/978-3-642-28869-2_2

Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. 2011. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*. ACM, New York, 487–498. DOI:http://dx.doi.org/10.1145/1926385.1926442

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. ACM, New York, 55–66. DOI:http://dx.doi.org/10.1145/1925844.1926394

Giovanni Bernardi and Alexey Gotsman. 2016. Robustness against consistency models with atomic visibility. In *CONCUR*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:15. DOI:http://dx.doi.org/10.4230/LIPIcs.CONCUR.2016.7

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and enforcing robustness against TSO. In *ESOP*. Springer-Verlag, Berlin, Heidelberg, 533–553. DOI:http://dx.doi.org/10.1007/978-3-642-37036-6_29

Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *POPL*. ACM, New York, 626–638. DOI:http://dx.doi.org/10.1145/3009837.3009888

Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding robustness against total store ordering. In *ICALP*. Springer, Berlin, Heidelberg, 428–440.

Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2018. Static serializability analysis for causal consistency. In *PLDI*. ACM, New York, 90–104. DOI:http://dx.doi.org/10.1145/3192366.3192415

Sebastian Burckhardt. 2014. Principles of eventual consistency. *Found. Trends Program. Lang.* 1, 1-2 (Oct. 2014), 1–150. DOI:http://dx.doi.org/10.1561/2500000011

Sebastian Burckhardt and Madanlal Musuvathi. 2008. Effective program verification for relaxed memory models. In *CAV*. Springer-Verlag, Berlin, Heidelberg, 107–120. DOI:http://dx.doi.org/10.1007/978-3-540-70545-1_12

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A framework for transactional consistency models with atomic visibility. In *26th International Conference* *on Concurrency Theory (CONCUR 2015) (LIPIcs)*, Vol. 42. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 58–71.

Egor Derevenetc. 2015. *Robustness against relaxed memory models*. Ph.D. Dissertation. University of Kaiserslautern. http://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/4074

Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2019. Verifying C11 programs operationally. In *PPoPP*. ACM, New York, 355–365. DOI:http://dx.doi.org/10.1145/3293883.3295702

Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. 2010. Parameterized memory models and concurrent separation logic. In *ESOP (LNCS)*, Vol. 6012. Springer, 267–286.

Phillip B. Gibbons and Ephraim Korach. 1997. Testing shared memories. *SIAM J. Comput.* 26, 4 (Aug. 1997), 1208–1244. DOI:http://dx.doi.org/10.1137/S0097539794279614

ISO/IEC 14882:2011. 2011. Programming Language C++. (2011).

ISO/IEC 9899:2011. 2011. Programming Language C. (2011).

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2017.17

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL 2017*. ACM, 175–189.

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. DOI:http://dx.doi.org/10.1145/3158105

Dexter Kozen. 1977. Lower bounds for natural proof systems. In *SFCS*. IEEE Computer Society, Washington, 254–266. DOI:http://dx.doi.org/10.1109/SFCS.1977.16

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL*. ACM, New York, 649–662. DOI:http://dx.doi.org/10.1145/2837614.2837643

Ori Lahav and Roy Margalit. 2019. Robustness against release/acquire semantics. In *PLDI (accepted for publication)*.

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries reasoning for weak memory models. In *ICALP*. Springer-Verlag, Berlin, Heidelberg, 311–323. DOI:http://dx.doi.org/10.1007/978-3-662-47666-6_25

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*. ACM, New York, 618–632. DOI:http://dx.doi.org/10.1145/3062341.3062352

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

Mapping 2016. C/C++11 mappings to processors. (2016). Retrieved June 27, 2018 from http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

Kartik Nagar and Suresh Jagannathan. 2018. Automated detection of serializability violations under weak consistency. In *CONCUR 2018*, Vol. 118. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 41:1–41:18. DOI:http://dx.doi.org/10.4230/LIPIcs.CONCUR.2018.41

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *TPHOLs*. Springer, Heidelberg, 391–407. DOI:http://dx.doi.org/10.1007/978-3-642-03359-9_27

Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6, 4 (1976), 319–340.

Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying read-copy-update in a logic for weak memory. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 110–120.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*. ACM, New York, 691–707. DOI:http://dx.doi.org/10.1145/2660193.2660243

Viktor Vafeiadis. 2017. Program verification under weak memory consistency using separation logic. In *CAV*. Springer International Publishing, Cham, 30–46.

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*. ACM, New York, 867–884. DOI:http://dx.doi.org/10.1145/2509136.2509532