Robustness Against Transactional Causal Consistency

Sidi Mohamed Beillahi
Université de Paris, IRIF, CNRS, F-75013 Paris, France
beillahi@irif.fr

Ahmed Bouajjani
Université de Paris, IRIF, CNRS, F-75013 Paris, France
abou@irif.fr

Constantin Enea
Université de Paris, IRIF, CNRS, F-75013 Paris, France
cenea@irif.fr

Abstract

Distributed storage systems and databases are widely used by various types of applications. Transactional access to these storage systems is an important abstraction allowing application programmers to consider blocks of actions (i.e., transactions) as executing atomically. For performance reasons, the consistency models implemented by modern databases are weaker than the standard serializability model, which corresponds to the atomicity abstraction of transactions executing over a sequentially consistent memory. Causal consistency for instance is one such model that is widely used in practice.

In this paper, we investigate application-specific relationships between several variations of causal consistency and we address the issue of verifying automatically if a given transactional program is robust against causal consistency, i.e., all its behaviors when executed over an arbitrary causally consistent database are serializable. We show that programs without write-write races have the same set of behaviors under all these variations, and we show that checking robustness is polynomial time reducible to a state reachability problem in transactional programs over a sequentially consistent shared memory. A surprising corollary of the latter result is that causal consistency variations which admit incomparable sets of behaviors admit comparable sets of robust programs. This reduction also opens the door to leveraging existing methods and tools for the verification of concurrent programs (assuming sequential consistency) for reasoning about programs running over causally consistent databases. Furthermore, it allows to establish that the problem of checking robustness is decidable when the programs executed at different sites are finite-state.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases Distributed Databases, Causal Consistency, Model Checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2019.30

Related Version Complete proofs can be found at `https://arxiv.org/abs/1906.12095`.

Funding This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

1 Introduction

Distribution and replication are widely adopted in order to implement storage systems and databases offering performant and available services. The implementations of these systems must ensure consistency guarantees allowing to reason about their behaviors in an abstract and simple way. Ideally, programmers of applications using such systems would like to have strong consistency guarantees, i.e., all updates occurring anywhere in the system are seen immediately and executed in the same order by all sites. Moreover, application programmers also need an abstract mechanism such as transactions, ensuring that blocks

```
                 t3 [x = 2
t1 [z = 1             r1 = z] //0
     x = 1]  ||  t4 [r2 = y   //1
t2 [y = 1]            r3 = x] //2
```

(a) `CCv` but not `CM`.

```
t1 [x = 1]             ||  t3 [x = 2]
t2 [r1 = x] //2            t4 [r2 = x] //1
```

(b) `CM` but not `CCv`.

```
                  t2 [x = 1]
t1 [x = 2]   ||   t3 [r1 = x] //2
                  t4 [r2 = x] //1
```

(c) `CC` but not `CM` nor `CCv`.

Figure 1 Computations showing the relation between `CC`, `CCv` and `CM`. Transactions are delimited using brackets, same site transactions are aligned vertically, and read values are given in comments.

of actions (writes and reads) of a site can be considered as executing atomically without interferences from actions of other sites. For transactional programs, the consistency model offering strong consistency is *serializability* [37], i.e., every computation of a program is equivalent to another one where transactions are executed serially one after another without interference. In the non-transactional case this model corresponds to *sequential consistency* (SC) [31]. However, while serializability and SC are easier to apprehend by application programmers, their enforcement (by storage systems implementors) requires the use of global synchronization between all sites, which is hard to achieve while ensuring availability and acceptable performances [24, 25]. For this reason, modern storage systems ensure weaker consistency guarantees. In this paper, we are interested in studying *causal consistency* [30].

Causal consistency is a fundamental consistency model implemented in several production databases, e.g., AntidoteDB, CockroachDB, and MongoDB, and extensively studied in the literature [7, 23, 33, 34, 39]. Basically, when defined at the level of actions, it guarantees that every two causally related actions, say a_1 is causally before (i.e., it has an influence on) a_2, are executed in that same order, i.e., a_1 before a_2, by all sites. The sets of updates visible to different sites may differ and read actions may return values that cannot be obtained in SC executions. The definition of causal consistency can be lifted to the level of transactions, assuming that transactions are visible to a site in their entirety (i.e., all their updates are visible at the same time), and they are executed by a site in isolation without interference from other transactions. In comparison to serializability, causal consistency allows that conflicting transactions, i.e., which read or write to a common location, be executed in different orders by different sites as long as they are not causally related. Actually, we consider three variations of causal consistency introduced in the literature, weak causal consistency (`CC`) [38, 14], causal memory (`CM`) [3, 38], and causal convergence (`CCv`) [18].

The weakest variation of causal consistency, namely `CC`, allows speculative executions and roll-backs of transactions which are not causally related (concurrent). For instance, the computation in Fig. 1c is only feasible under `CC`: the site on the right applies `t2` after `t1` before executing `t3` and roll-backs `t2` before executing `t4`. `CCv` and `CM` offer more guarantees. `CCv` enforces a total *arbitration* order between all transactions which defines the order in which delivered concurrent transactions are executed by *every* site. This guarantees that all sites reach the same state when all transactions are delivered. `CM` ensures that *all* values read by a site can be explained by an interleaving of transactions consistent with the causal order, enforcing thus PRAM consistency [32] on top of `CC`. Contrary to `CCv`, `CM` allows that two sites diverge on the ordering of concurrent transactions, but both models do not allow roll-backs of concurrent transactions. Thus, `CCv` and `CM` are incomparable in terms of computations they admit. The computation in Fig. 1a is not admitted by `CM` because there is no interleaving of those transactions that explains the values read by the site on the right: reading 0 from `z` implies that the transactions on the left must be applied after `t3` while reading 1 from `y` implies that both `t1` and `t2` are applied before `t4` which contradicts reading 2 from `x`. However, this computation is possible under `CCv` because `t1` can be delivered to the right after executing `t3` but arbitrated before `t3`, which implies that the write to `x` in `t1` will be lost. The `CM` computation in Fig. 1b is not possible under `CCv` because there is no arbitration order that could explain both reads from `x`.

As a first contribution of our paper, we show that the three causal consistency semantics coincide for transactional programs containing no *write-write races*, i.e., concurrent transactions writing on a common variable. We also show that if a transactional program has a write-write race under one of these semantics, then it must have a write-write race under any of the other two semantics. This property is rather counter-intuitive since `CC` is strictly weaker than both `CCv` and `CM`, and `CCv` and `CM` are incomparable (in terms of admitted behaviors). Notice that each of the computations in Figures 1b, 1a, and 1c contains a write-write race which explains why none of these computations is possible under all three semantics.

Then, we investigate the problem of checking *robustness* of application programs against causal consistency relaxations: Given a program P and a causal consistency variation X, we say that P is robust against X if the set of computations of P when running under X is the same as its set of computations when running under *serializability*. This means that it is possible to reason about the behaviors of P assuming the simpler serializability model and no additional synchronization is required when P runs under X such that it maintains all the properties satisfied under serializability. Checking robustness is not trivial, it can be seen as a form of checking program equivalence. However, the equivalence to check is between two versions of the same program, obtained using two different semantics, one more permissive than the other one. The goal is to check that this permissiveness has actually no effect on the particular program under consideration. The difficulty in checking robustness is to apprehend the extra behaviors due to the reorderings introduced by the relaxed consistency model w.r.t. serializability. This requires a priori reasoning about complex order constraints between operations in arbitrarily long computations, which may need maintaining unbounded ordered structures, and make the problem of checking robustness hard or even undecidable.

We show that verifying robustness of transactional programs against causal consistency can be reduced in polynomial time to the reachability problem in concurrent programs over SC. This allows to reason about distributed applications running on causally consistent storage systems using the existing verification technology and it implies that the robustness problem is decidable for finite-state programs; the problem is PSPACE-complete when the number of sites is fixed, and EXPSPACE-complete otherwise. This is the first result on the decidability and complexity of verifying robustness against causal consistency. In fact, the problem of verifying robustness has been considered in the literature for several consistency models of distributed systems, including causal consistency [12, 16, 17, 20, 35]. These works provide (over- or under-)approximate analyses for checking robustness, but none of them provides precise (sound and complete) algorithmic verification methods for solving this problem, nor addresses its decidability and complexity.

The approach we adopt for tackling this verification problem is based on a precise characterization of the set of robustness violations, i.e., executions that are causally consistent but not serializable. For both `CCv` and `CM`, we show that it is sufficient to search for a special type of robustness violations, that can be simulated by serial (SC) computations of an instrumentation of the original program. These computations maintain the information needed to recognize the pattern of a violation that would have occurred in the original program under a causally consistent semantics (executing the same set of operations). A surprising consequence of these results is that a program is robust against `CM` iff it is robust against `CC`, and robustness against `CM` implies robustness against `CCv`. This shows that the causal consistency variations we investigate can be incomparable in terms of the admitted behaviors, but comparable in terms of the robust applications they support.

2 Causal Consistency

Program syntax. We consider a simple programming language where a program is parallel composition of *processes* distinguished using a set of identifiers $\mathbb{P}$. Each process is a sequence of *transactions* and each transaction is a sequence of *labeled instructions*. Each transaction starts with a `begin` instruction and finishes with an `end` instruction. Each other instruction is either an assignment to a process-local *register* from a set $\mathbb{R}$ or to a *shared variable* from a set $\mathbb{V}$, or an `assume` statement. The assignments use values from a data domain $\mathbb{D}$. An assignment to a register $\langle reg \rangle := \langle var \rangle$ is called a *read* of $\langle var \rangle$ and an assignment to a shared variable $\langle var \rangle := \langle reg\text{-}expr \rangle$ is called a *write* to $\langle var \rangle$ ($\langle reg\text{-}expr \rangle$ is an expression over registers). The statement `assume` $\langle bexpr \rangle$ blocks the process if the Boolean expression $\langle bexpr \rangle$ over registers is false. Each instruction is followed by a `goto` statement which defines the evolution of the program counter. Multiple instructions can be associated with the same label which allows us to write non-deterministic programs and multiple `goto` statements can direct the control to the same label which allows us to mimic imperative constructs like loops and conditionals. We assume that the control cannot pass from one transaction to another without going as expected through `begin` and `end` instructions.

Causal memory (CM) semantics. Informally, the semantics of a program under causal memory is defined as follows. The shared variables are replicated across each process, each process maintaining its own local valuation of these variables. During the execution of a transaction in a process, the shared-variable writes are stored in a *transaction log* which is visible only to the process executing the transaction and which is broadcasted to all the other processes at the end of the transaction[1]. To read a shared variable x, a process p first accesses its transaction log and takes the last written value on x, if any, and then its own valuation of the shared variables, if x was not written during the current transaction. Transaction logs are delivered to every process in an order consistent with the *causal delivery* relation between transactions, i.e., the transitive closure of the union of the *program order* (the order in which transactions are executed by a process), and the *delivered-before* relation (a transaction t_1 is delivered-before a transaction t_2 iff the log of t_2 has been delivered at the process executing t_1 before t_1 starts). By an abuse of terminology, we call this property *causal delivery*. Once a transaction log is delivered, it is immediately applied on the shared-variable valuation of the receiving process. Also, no transaction log can be delivered to a process p while p is executing another transaction, we call this property *transaction isolation*.

Causal convergence (CCv) semantics. Compared to causal memory, causal convergence ensures eventual consistency of process-local copies of the shared variables. Each transaction log is associated with a timestamp and a process applies a write on some variable x from a transaction log only if it has a timestamp larger than the timestamps of all the transaction logs it has already applied and that wrote the same variable x. For simplicity, we assume that the transaction identifiers play the role of timestamps, which are totally ordered according to some relation $<$. `CCv` satisfies both *causal delivery* and *transaction isolation* as well.

Weak causal consistency (CC) semantics. Compared to the previous semantics, `CC` allows that reads of the same process observe concurrent writes as executing in different orders. Each process maintains a *set* of values for each shared variable, and a read returns any one

[1] For simplicity, we assume that every transaction commits. The effects of aborted transactions shouldn't be visible to any process.

$$\begin{array}{l}\mathsf{begin}(p1,t1)\\ \quad\mathsf{isu}(p1,t1,x,1)\\ \mathsf{end}(p1,t1)\end{array} \cdot \begin{array}{l}\mathsf{begin}(p2,t3)\\ \quad\mathsf{isu}(p2,t3,x,2)\\ \mathsf{end}(p2,t3)\end{array} \cdot \mathsf{del}(p1,t3) \cdot \mathsf{del}(p2,t1) \cdot \begin{array}{l}\mathsf{begin}(p2,t4)\\ \quad\mathsf{ld}(p2,t4,x,1)\\ \mathsf{end}(p2,t4)\end{array} \cdot \begin{array}{l}\mathsf{begin}(p1,t2)\\ \quad\mathsf{ld}(p1,t2,x,2)\\ \mathsf{end}(p1,t2)\end{array}$$

(a) CM execution of the program in Fig. 1b.

$$\begin{array}{l}\mathsf{begin}(p1,t1)\\ \quad\mathsf{isu}(p1,t1,z,1)\\ \quad\mathsf{isu}(p1,t1,x,1)\\ \mathsf{end}(p1,t1)\end{array} \cdot \begin{array}{l}\mathsf{begin}(p2,t3)\\ \quad\mathsf{isu}(p2,t3,x,2)\\ \quad\mathsf{ld}(p2,t3,z,0)\\ \mathsf{end}(p2,t3)\end{array} \cdot \mathsf{del}(p1,t3) \cdot \mathsf{del}(p2,t1) \cdot \begin{array}{l}\mathsf{begin}(p1,t2)\\ \quad\mathsf{isu}(p1,t2,y,1)\\ \mathsf{end}(p1,t2)\end{array} \cdot \mathsf{del}(p2,t2) \cdot \begin{array}{l}\mathsf{begin}(p2,t4)\\ \quad\mathsf{ld}(p2,t4,y,1)\\ \quad\mathsf{ld}(p2,t4,x,2)\\ \mathsf{end}(p2,t4)\end{array}$$

(b) CCv execution of the program in Figure 1a.

$$\begin{array}{l}\mathsf{begin}(p1,t1)\\ \quad\mathsf{isu}(p1,t1,x,2)\\ \mathsf{end}(p1,t1)\end{array} \cdot \begin{array}{l}\mathsf{begin}(p2,t2)\\ \quad\mathsf{isu}(p2,t2,x,1)\\ \mathsf{end}(p2,t2)\end{array} \cdot \mathsf{del}(p2,t1) \cdot \begin{array}{l}\mathsf{begin}(p2,t3)\\ \quad\mathsf{ld}(p2,t3,x,2)\\ \mathsf{end}(p2,t3)\end{array} \cdot \begin{array}{l}\mathsf{begin}(p2,t4)\\ \quad\mathsf{ld}(p2,t4,x,1)\\ \mathsf{end}(p2,t4)\end{array} \cdot \mathsf{del}(p1,t2)$$

(c) CC execution of the program in Figure 1c.

Figure 2 For readability, the sub-sequences of events delimited by begin and end are aligned vertically, the execution-flow advancing from left to right and top to bottom.

of these values non-deterministically. Transaction logs are associated with *vector clocks* [30] which represent the causal delivery relation, i.e., a transaction t_1 is before t_2 in causal-delivery iff the vector clock of t_1 is smaller than the vector clock of t_2. We assume that transactions identifiers play the role of vector clocks, which are partially ordered according to some relation $<$. When applying a transaction log on the shared-variable valuation of the receiving process, we only keep the values that were written by *concurrent* transactions (not related by causal delivery). CC satisfies both *causal delivery* and *transaction isolation*.

Program execution. The semantics of a program $\mathcal{P}$ under a causal consistency semantics $\mathsf{X} \in \{\mathtt{CCv}, \mathtt{CM}, \mathtt{CC}\}$ is defined using a labeled transition system $[\mathcal{P}]_{\mathsf{X}}$ where the set $\mathbb{E}\mathsf{v}$ of transition labels, called *events*, is defined by:

$$\mathbb{E}\mathsf{v} = \{\mathsf{begin}(p,t), \mathsf{ld}(p,t,x,v), \mathsf{isu}(p,t,x,v), \mathsf{del}(p,t), \mathsf{end}(p,t) : p \in \mathbb{P}, t \in \mathbb{T}, x \in \mathbb{V}, v \in \mathbb{D}\}$$

where begin and end label transitions corresponding to the start, resp., the end of a transaction, isu and ld label transitions corresponding to writing, resp., reading, a shared variable during some transaction, and del labels transitions corresponding to applying a transition log received from another process on the local copy of the shared variables. An event isu is called an *issue* while an event del is called a *store*. An execution of $[\mathcal{P}]_{\mathsf{X}}$ is a sequence of events $\rho = ev_1 \cdot ev_2 \cdot \ldots$ labeling the transitions. Fig. 2a shows an execution under CM. This satisfies transaction isolation since no transaction is delivered while another transaction is executing. The execution in Fig. 2a is not possible under CCv since $t4$ and $t2$ read 2 and 1 from x, respectively. This is possible only if $t1$ and $t3$ write x at $p2$ and $p1$, respectively, which contradicts the definition of CCv where we cannot have both $t1 < t3$ and $t3 < t1$. Fig. 2b shows an execution under CCv (we assume $t_1 < t_2 < t_3 < t_4$). Notice that $\mathsf{del}(p2,t1)$ did not result in an update of x because the timestamp t_1 is smaller than the timestamp of the last transaction that wrote x at p_2, namely t_3, a behavior that is not possible under CM. The two processes converge and store the same shared variable copy at the end of the execution. Fig. 2c shows an execution under CC, which is not possible under CCv and CM because $t3$ and $t4$ read 2 and 1, respectively. Since the transactions t_1 and t_2 are concurrent, p_2 stores both values 2 and 1 written by these transactions. A read of x can return any of these two values.

Execution summary. Let ρ be an execution under $\mathsf{X} \in \{\mathtt{CCv}, \mathtt{CM}, \mathtt{CC}\}$, a sequence τ of events $\mathsf{isu}(p,t)$ and $\mathsf{del}(p,t)$ with $p \in \mathbb{P}$ and $t \in \mathbb{T}$ is called a *summary of* ρ if it is obtained from ρ by substituting every sub-sequence of transitions in ρ delimited by a begin and an end transition, with a single "macro-event" $\mathsf{isu}(p, t)$. For example, $\mathsf{isu}(p1, t1) \cdot \mathsf{isu}(p2, t3) \cdot \mathsf{del}(p1, t3) \cdot \mathsf{del}(p2, t1) \cdot \mathsf{isu}(p2, t4) \cdot \mathsf{isu}(p1, t2)$ is a summary of the execution in Fig. 2a.

We say that a transaction t in ρ performs an *external read* of a variable x if ρ contains an event $\mathsf{ld}(p, t, x, v)$ which is not preceded by a write on x of t (i.e., $\mathsf{isu}(p, t, x, v)$). Under CM and CC, a transaction t *writes* a variable x if ρ contains $\mathsf{isu}(p, t, x, v)$, for some v. In Fig. 2a, $t2$ and $t4$ perform external reads, and $t2$ writes to y. A transaction t executed by a process p *writes x at process $p' \neq p$* if t writes x and ρ contains $\mathsf{del}(p', t)$ (e.g., in Fig. 2a, $t1$ writes x at $p2$). Under CCv, we say that a transaction t executed by a process p *writes x at process $p' \neq p$* if t writes x and ρ contains an event $\mathsf{del}(p', t)$ which is not preceded by an event $\mathsf{del}(p', t')$ (or $\mathsf{isu}(p', t')$) with $t < t'$ and t' writing x (if it would be preceded by such an event then the write to x of t will be discarded). For example, in Fig. 2b, $t1$ does *not* write x at $p2$.

3 Write-Write Race Freedom

We say that an execution ρ has a *write-write race* on a shared variable x if there exist two concurrent transactions t_1 and t_2 that were issued in ρ and each transaction contains a write to the variable x. We call ρ write-write race free if there is no variable x such that ρ has a write-write race on x. Also, we say a program $\mathcal{P}$ is *write-write race free* under a consistency semantics $\mathsf{X} \in \{\mathtt{CCv}, \mathtt{CM}, \mathtt{CC}\}$ iff for every $\rho \in \mathbb{E}\mathsf{x}_{\mathsf{X}}(\mathcal{P})$, ρ is write-write race free.

We show that if a given program has a write-write race under one of the three causal consistency semantics then it must have a write-write race under the remaining two. The intuition behind this is that the three semantics coincide for programs without write-write races. Indeed, without concurrent transactions that write to the same variable, every process local valuation of a shared variable will be a singleton set under CC and no process will ever discard a write when applying an incoming transaction log under CCv.

▶ **Theorem 1.** *Given a program $\mathcal{P}$ and two consistency semantics $X, Y \in \{\mathtt{CCv}, \mathtt{CM}, \mathtt{CC}\}$, $\mathcal{P}$ has a write-write race under X iff $\mathcal{P}$ has a write-write race under Y.*

The following result shows that indeed, the three causal consistency semantics coincide for programs which are write-write race free under any one of these three semantics.

▶ **Theorem 2.** *Let $\mathcal{P}$ be a program. Then, $\mathbb{E}\mathsf{x}_{\mathtt{CC}}(\mathcal{P}) = \mathbb{E}\mathsf{x}_{\mathtt{CCv}}(\mathcal{P}) = \mathbb{E}\mathsf{x}_{\mathtt{CM}}(\mathcal{P})$ iff $\mathcal{P}$ has no write-write race under CM, CCv, or CC.*

4 Programs Robustness

4.1 Program Traces

We define an abstraction of executions satisfying transaction isolation, called *trace*. Intuitively, a trace forgets the order in which shared-variables are accessed inside a transaction and the order between transactions accessing different variables. The trace of an execution ρ is obtained by adding several standard relations between events in its summary which record the data-flow, e.g. which transaction wrote the value read by another transaction.

The *trace* of an execution ρ is a tuple $\mathsf{tr}(\rho) = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ where τ is the summary of ρ, PO is the *program order*, which relates any two issue events $\mathsf{isu}(p, t)$ and $\mathsf{isu}(p, t')$ that occur in this order in τ, WR is the *write-read* relation (also called *read-from*), which relates events of two transactions t and t' such that t writes a value that t' reads, WW is the *write-write* order (also called store-order), which relates events of two transactions that write to the same variable, and RW is the *read-write* relation (also called *conflict*), which relates events of two transactions t and t' such that t reads a value overwritten by t', and STO is the *same-transaction* relation, which relates events of the same transaction.

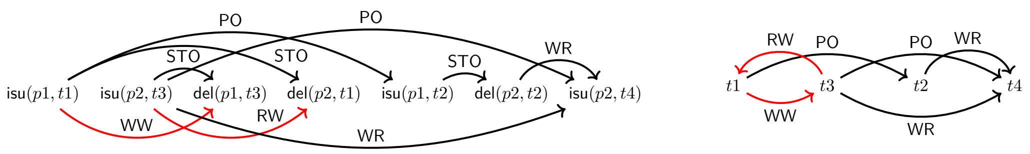


Figure 3 The trace of the execution in Fig. 2b and its transactional happens-before.

Formally, WR relates any two events $ev_1 \in \{\mathsf{isu}(p,t), \mathsf{del}(p,t)\}$ and $ev_2 = \mathsf{isu}(p,t')$ that occur in this order in τ such that t' performs an external read of x, and ev_1 is the last event in τ before ev_2 such that t writes x if $ev_1 = \mathsf{isu}(p,t)$, and t writes x at p if $ev_1 = \mathsf{del}(p,t)$ (we may use $\mathsf{WR}(x)$ to emphasize the variable x). Also, WW relates any two events $ev_1 \in \{\mathsf{isu}(p,t_1), \mathsf{del}(p,t_1)\}$ and $ev_2 \in \{\mathsf{isu}(p,t_2), \mathsf{del}(p,t_2)\}$ that occur in this order in τ provided that t_1 and t_2 both write the same variable x, and if ρ is an execution under causal convergence, then (1) if $ev_i = \mathsf{del}(p,t_i)$, for some $i \in \{1,2\}$, then t_i writes x at p, and (2) if $ev_1 \in \{\mathsf{isu}(p,t_1), \mathsf{del}(p,t_1)\}$ and $ev_2 = \mathsf{del}(p,t_2)$, then $t_1 < t_2$ (we may use $\mathsf{WW}(x)$ to emphasize the variable x). We also define $\mathsf{RW}(x) = \mathsf{WR}^{-1}(x);\mathsf{WW}(x)$ (we use ; to denote the standard composition of relations) and $\mathsf{RW} = \bigcup_{x \in \mathbb{V}} \mathsf{RW}(x)$. If a transaction t reads the initial value of x then $\mathsf{RW}(x)$ relates $\mathsf{isu}(p,t)$ to $\mathsf{isu}(p',t')$ of any other transaction t' which writes to x (i.e., $(\mathsf{isu}(p,t), \mathsf{isu}(p',t')) \in \mathsf{RW}(x)$). Finally, STO relates any event $\mathsf{isu}(p,t)$ with the set of events $\mathsf{del}(p',t)$, where $p \neq p'$.

Then, we define the *happens-before* relation HB as the transitive closure of the union of all the relations in the trace, i.e., $\mathsf{HB} = (\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW} \cup \mathsf{STO})^+$. Since we reason about only one trace at a time, we may say that a trace is simply a summary τ, keeping the relations implicit. The trace of the CCv execution in Fig. 2b is shown on the left of Fig. 3. $\mathbb{T}\mathsf{r}(\mathcal{P})_\mathsf{X}$ denotes the set of traces of executions of a program $\mathcal{P}$ under $\mathsf{X} \in \{\mathtt{CCv},\ \mathtt{CM},\ \mathtt{CC}\}$.

The *causal order* CO of a trace $tr = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ is the transitive closure of the union of the program order, write-read relation, and the same-transaction relation, i.e., $\mathsf{CO} = (\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{STO})^+$. For readability, we write $ev_1 \to_{\mathsf{HB}} ev_2$ instead of $(ev_1, ev_2) \in \mathsf{HB}$.

4.2 Program Semantics Under Serializability

The semantics of a program under serializability [37] can be defined using a transition system where the configurations keep a single shared-variable valuation (accessed by all processes) with the standard interpretation of read or write statements. Each transaction executes in isolation. Alternatively, the serializability semantics can be defined as a restriction of $[\mathcal{P}]_\mathsf{X}$, $\mathsf{X} \in \{\mathtt{CCv},\ \mathtt{CM},\ \mathtt{CC}\}$, to the set of executions where each transaction is *immediately* delivered to all processes, i.e., each event $\mathsf{end}(p,t)$ is immediately followed by all $\mathsf{del}(p',t)$ with $p' \neq p$. Such executions are called *serializable* and the set of serializable executions of a program $\mathcal{P}$ is denoted by $\mathbb{E}\mathsf{x}_{\mathtt{SER}}(\mathcal{P})$. The latter definition is easier to reason about when relating executions under causal consistency and serializability, respectively.

A trace tr is called *serializable* if it is the trace of a serializable execution. Let $\mathbb{T}\mathsf{r}_{\mathtt{SER}}(\mathcal{P})$ denote the set of serializable traces. Given a serializable trace $tr = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ we have that every event $\mathsf{isu}(p,t)$ in τ is immediately followed by all $\mathsf{del}(p',t)$ with $p' \neq p$. For simplicity, we write τ as a sequence of "atomic macro-events" (p,t) where (p,t) denotes a sequence $\mathsf{isu}(p,t) \cdot \mathsf{del}(p_1,t) \cdot \ldots \cdot \mathsf{del}(p_n,t)$ for some $p \in \mathbb{P} = \{p, p_1, \ldots, p_n\}$. We say that t is "atomic". In Fig. 3, $t2$ is atomic and we can use $(p2, t3)$ instead of $\mathsf{isu}(p2,t3)\mathsf{del}(p1,t3)$.

Since multiple executions may have the same trace, it is possible that an execution ρ produced by a variation of causal consistency has a serializable trace $\mathsf{tr}(\rho)$ even though

```
    p1                         p2
t1 [r1 = x    //0   ||   t2 [r2 = x    //0
    x  = r1 + 1]             x = r2 + 1]
```

(a) Lost Update (LU).

```
    p1                     p2
t1 [x = 1]          ||   t3 [y = 1]
t2 [r1 = y] //0          t4 [r2 = x] //0
```

(b) Store Buffering (SB).

```
[a = 1        if (a == 1)
 z = 1          [x = 2
 x = 1    ||     r1 = z  //0
 y = 1]          r2 = y  //1
                 r3 = x] //2
```

(c) Without transactions, non-robust against CCv.

```
[a = 1            if (a == 1)
 x = 1      ||      [x = 2
 r1 = x] //2         r2 = x] //1
```

(d) Without transactions, non-robust only against CM.

```
if ( * )          if ( * )
  [x = 1]    ||     [x = 2]
else              else
  [r1 = x]          [r2 = x]
```

(e) Robust against both CM and CCv.

```
               [r2 = x
[x = 1]   ||    if (r2 == 1)
[r1 = y]         y = 1]
```

(f) Robust against both CM and CCv.

Figure 4 (Non-)robust programs. For non-robust programs, the read instructions are commented with the values they return in robustness violations. The condition of `if-else` is checked inside a transaction whose demarcation is omitted for readability (`*` denotes non-deterministic choice).

end(p, t) actions may not be immediately followed by del(p', t) actions. However, ρ would be equivalent, up to reordering of "independent" (or commutative) transitions, to a serializable execution. The happens-before relation between events is extended to transactions as follows: a transaction t_1 *happens-before* another transaction $t_2 \neq t_1$ if the trace *tr* contains an event of transaction t_1 which happens-before an event of t_2. The happens-before relation between transactions is denoted by HB_t and called *transactional happens-before* (an example is given on the right of Fig. 3). The following result characterizes serializable traces.

▶ **Theorem 3** ([2, 41]). *A trace tr is serializable iff* HB_t *is acyclic.*

4.3 Robustness Problem

We consider the problem of checking whether the causally-consistent semantics of a program produces the same set of traces as the serializability semantics.

▶ **Definition 4.** *A program $\mathcal{P}$ is called* robust *against a semantics $X \in \{\texttt{CCv}, \texttt{CM}, \texttt{CC}\}$ iff $\mathbb{T}r_X(\mathcal{P}) = \mathbb{T}r_{\texttt{SER}}(\mathcal{P})$.*

Since $\mathbb{T}r_{\texttt{SER}}(\mathcal{P}) \subseteq \mathbb{T}r_{\mathsf{X}}(\mathcal{P})$, the problem of checking robustness of a program $\mathcal{P}$ against a semantics X boils down to checking whether there exists a trace $tr \in \mathbb{T}r_{\mathsf{X}}(\mathcal{P}) \setminus \mathbb{T}r_{\texttt{SER}}(\mathcal{P})$. We call *tr* a *robustness violation* (or *violation*, for short). By Theorem 3, HB_t of *tr* is cyclic.

We discuss several examples of programs which are (non-) robust against both CM and CCv or only one of them. Fig. 4a and Fig. 4b show examples of programs that are *not* robust against both CM and CCv, which have also been discussed in the literature on weak memory models, e.g. [6]. The execution of Lost Update under both CM and CCv allows that the two reads of x in transactions $t1$ and $t2$ return 0 although this cannot happen under serializability. Also, executing Store Buffering under both CM and CCv allows that the reads of x and y return 0 although this would not be possible under serializability. These values are possible because the first transaction in each of the processes may not be delivered to the other process.

Assuming for the moment that each instruction in Fig. 4c and Fig. 4d forms a different transaction, the values we give in comments show that the program in Fig. 4c, resp., Fig. 4d, is not robust against CCv, resp., CM. The values in Fig. 4c are possible assuming that the timestamp of the transaction `[x = 1]` is smaller than the timestamp of `[x = 2]` (which means that if the former is delivered after the second process executes `[x = 2]`, then it will be discarded). Moreover, enlarging the transactions as shown in Fig. 4c, the program becomes robust against CCv. The values in Fig. 4d are possible under CM because different processes do not need to agree on the order in which to apply transactions, each process applying the transaction received from the other process last. However, under CCv this behavior is not possible, the program being actually robust against CCv. As in the previous case, enlarging the transactions as shown in the figure leads to a robust program against CM.

We end the discussion with several examples of programs that are robust against both CM and CCv. These are simplified models of real applications reported in [28]. The program in Fig. 4e can be understood as the parallel execution of two processes that either create a new user of some service, represented abstractly as a write on a variable x or check its credentials, represented as a read of x (the non-deterministic choice abstracts some code that checks whether the user exists). Clearly this program is robust against both CM and CCv since each process does a single access to the shared variable. Although we considered simple transactions that access a single shared-variable this would hold even for "bigger" transactions that access an arbitrary number of variables. The program in Fig. 4f can be thought of as a process creating a new user of some service and reading some additional data in parallel to a process that updates that data only if the user exists. It is rather easy to see that it is also robust against both CM and CCv.

5 Robustness Against Causal Consistency

We consider now the issue of checking robustness against a variation of causal consistency, considering first the case of CCv and CM. The result concerning CC is derived from the one concerning CM. We show next that a robustness violation should contain at least an issue and a store event of the same transaction that are separated by another event that occurs after the issue and before the store and which is related to both via the happens-before relation. Otherwise, since any two events which are not related by happens-before could be swapped in order to derive an execution with the same trace, every store event could be swapped until it immediately follows the corresponding issue and the execution would be serializable.

▶ **Lemma 5.** *Given a robustness violation τ, there exists a transaction t such that $\tau = \alpha \cdot \mathit{isu}(p,t) \cdot \beta \cdot \mathit{del}(p_0,t) \cdot \gamma$ and an event $a \in \beta$ such that $(\mathit{isu}(p,t), a) \in \mathsf{HB}$ and $(a, \mathit{del}(p_0,t)) \in \mathsf{HB}$.*

The transaction t in the trace τ above is called a *delayed* transaction. The happens-before constraints imply that t belongs to an HB_t cycle.

Next, we show that a program which is not robust against CCv or CM admits violations of particular shapes, which enables reducing robustness checking to a reachability problem in a program running under serializability (presented in Section 6).

5.1 Robustness Violations under Causal Convergence

Roughly, our characterization of CCv robustness violations states that the first delayed transaction (which must exist by Lemma 5) is followed by a possibly-empty sequence of delayed transactions that form a "causality chain", i.e., every new delayed transaction is causally ordered after the previous delayed transaction. Moreover, the issue event of the last delayed transaction happens-before the issue event of another transaction that reads a variable updated by the first delayed transaction (this completes a cycle in HB_t). We say that a sequence of issue events $ev_1 \cdot ev_2 \cdot \ldots ev_n$ forms a *causality chain* when $(ev_i, ev_{i+1}) \in \mathsf{CO}$ for all $1 \leq i \leq n-1$. Also, for simplicity, we use "macro-events" (p,t) even in traces obtained under causal consistency (recall that this notation was introduced to simplify serializable traces),

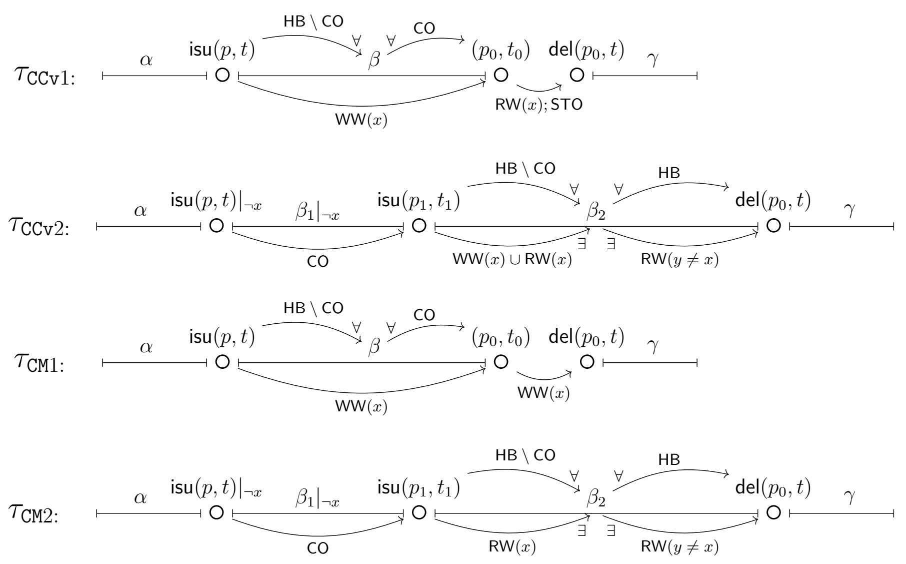


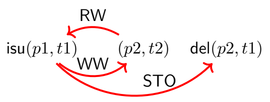
Figure 5 Robustness violation patterns. We use $a \xrightarrow{R} {}^{\forall} \beta$ to denote $\forall\, b \in \beta.\ (a,b) \in R$. We use $\beta_1|_{\neg x}$ to say that all delayed transactions in β_1 do not access x. For violations $\tau_{\texttt{CCv1}}$ and $\tau_{\texttt{CM1}}$, t is the only delayed transaction. For $\tau_{\texttt{CCv2}}$ and $\tau_{\texttt{CM2}}$, all delayed transactions are in $\beta_1|_{\neg x}$ and they form a causality chain that starts at isu(p,t) and ends at isu(p_1,t_1).

i.e., we assume that any sequence of events formed of an issue isu(p,t) followed immediately by all the store events del(p',t) is replaced by (p,t). Then, all the relations that held between an event ev of such a sequence and another event ev', e.g., $(ev, ev') \in$ PO, are defined to hold as well between the corresponding macro-event (p,t) and ev', e.g, $((p,t), ev') \in$ PO.

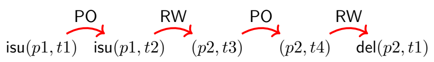
▶ **Theorem 6.** *A program $\mathcal{P}$ is* not *robust against* CCv *iff $\mathbb{T}r(\mathcal{P})_{\texttt{CCv}}$ contains a trace of type $\tau_{\texttt{CCv1}} = \alpha \cdot isu(p,t) \cdot \beta \cdot (p_0,t_0) \cdot del(p_0,t) \cdot \gamma$ or $\tau_{\texttt{CCv2}} = \alpha \cdot isu(p,t) \cdot \beta_1 \cdot isu(p_1,t_1) \cdot \beta_2 \cdot del(p_0,t) \cdot \gamma$ that satisfies the properties given in Fig. 5.*

Above, $\tau_{\texttt{CCv1}}$ contains a single delayed transaction while $\tau_{\texttt{CCv2}}$ may contain arbitrarily many delayed transactions. The issue event of the last delayed transactions, i.e., isu(p,t) in $\tau_{\texttt{CCv1}}$ and isu(p_1,t_1) in $\tau_{\texttt{CCv2}}$, happens before (p_0,t_0) and some event in β_2, respectively, which read a variable updated by the first delayed transaction. The theorem above allows $\beta_1 = \epsilon$, $\beta_2 = \epsilon$, $\beta = \epsilon$, $\gamma = \epsilon$, $p = p_1$, $t = t_1$, and t_1 to be a read-only transaction. If t_1 is a read-only transaction then isu(p_1,t') has the same effect as (p_1,t_1) since t_1 does not contain writes. Fig. 6a and Fig. 6b show two violations under CCv.

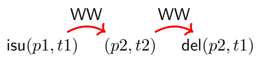
The violation patterns in Theorem 6 characterize *minimal* robustness violations where the measure defined as the sum of the distances (number of events that are causally related to the issue) between issue and store events of the same transaction is minimal. Minimality enforces the constraints stated above. For example, in the context of $\tau_{\texttt{CCv2}}$, the delayed transactions in β_1 cannot create a cycle in the transactional happens-before (otherwise, $\alpha \cdot$ isu$(p,t) \cdot \beta_1 \cdot$ del$(p_0,t) \cdot \gamma'$ would be a violation with a smaller measure, which contradicts minimality). Also, if it were to have a delayed transaction t_2 in β_2 (resp., β for $\tau_{\texttt{CCv1}}$), then it is possible to remove some transaction (all its issue and store events) from the original trace and obtain a new violation with a smaller measure. For instance, in the case of β_2, if $t \neq t_1$, then we can safely remove the last delayed transaction (i.e., t_1), that is causally dependent on the first delayed transaction, since all events in $\beta_2 \cdot$ del$(p_0,t) \cdot \gamma$ neither read from the



(a) Violation of LU program in Fig. 4a.



(b) Violation of SB program in Fig. 4b.



(c) Violation of LU program in Fig. 4a.

Figure 6 (a) A $\tau_{\texttt{CCv1}}$ violation where $\beta_2 = \epsilon$, $\gamma = \epsilon$, and t and t_0 correspond to $t1$ and $t2$. (b) A $\tau_{\texttt{CCv2}}$ (resp., $\tau_{\texttt{CM2}}$) violation where t and t_1 correspond to $t1$ and $t2$. t_1 is a read-only transaction. Also, $\beta_1 = \epsilon$, $\beta_2 = (p2, t3) \cdot (p2, t4)$, $\gamma = \epsilon$, such that $(\mathsf{isu}(p1, t2), (p2, t3)) \in \mathsf{RW}(y)$ and $((p2, t4), \mathsf{del}(p2, t1)) \in \mathsf{RW}(x)$. (c) A $\tau_{\texttt{CM1}}$ violation with $\beta_2 = \gamma = \epsilon$, and t and t_0 correspond to $t1$ and $t2$. In all traces, we show only the relations that are part of the happens-before cycle.

writes of t_1 nor are issued by the same process as t_1. The resulting trace is still a robustness violation (because of the HB_t cycle involving t_2) but with a smaller measure. Note that all processes that delayed transactions, stop executing new transactions in β_2 (resp., β) because of the relation $\mathsf{HB} \setminus \mathsf{CO}$, shown in Fig. 5, between the delayed transaction t_1 (resp., t) and events in β_2 (resp., β). This characterization of minimal violations is essential for deriving an optimal reduction of robustness checking to SC reachability (presented in Section 6).

5.2 Robustness Violations under Causal Memory

The characterization of robustness violations under `CM` is at some level similar to that of robustness violations under `CCv`. However, some instance of the violation pattern under `CCv` is not possible under `CM` and `CM` admits some class of violations that is not possible under `CCv`. This reflects the fact that these consistency models are incomparable in general. The following theorem gives the precise characterization. Roughly, a program is not robust if it admits a violation which can either be because of two concurrent transactions that write to the same variable (a write-write race) or because of a restriction of the pattern admitted by `CCv` where the last delayed transaction must be related only by RW in a happens-before path with future transactions. The first pattern is not admitted by `CCv` because the writes to each variable are executed according to the timestamp order.

▶ **Theorem 7.** *A program $\mathcal{P}$ is not robust against* `CM` *iff $\mathbb{T}r(\mathcal{P})_{\texttt{CM}}$ contains a trace of type $\tau_{\texttt{CM1}} = \alpha \cdot isu(p, t) \cdot \beta \cdot (p_0, t_0) \cdot del(p_0, t) \cdot \gamma$ or $\tau_{\texttt{CM2}} = \alpha \cdot isu(p, t) \cdot \beta_1 \cdot isu(p_1, t_1) \cdot \beta_2 \cdot del(p_0, t) \cdot \gamma$ that satisfies the properties given in Fig. 5.*

The `CM` violation in Fig. 6b (pattern $\tau_{\texttt{CM2}}$) is a violation under `CCv` as well which corresponds to the pattern $\tau_{\texttt{CCv2}}$. The detection of the violation pattern $\tau_{\texttt{CM1}}$ (e.g., Fig. 6c) implies the existence of a write-write race under `CM`. Conversely, if a program has a trace which contains a write-write race under `CM`, then this trace must be a robustness violation since the two transactions, that caused the write-write race, form a cycle in the store order hence a cycle in the transactional happens-before order[2]. Thus, the program is not robust against `CM`. Therefore, a program which is robust against `CM` is also write-write race free under `CM`. Since without write-write races, the `CM` and the `CCv` semantics coincide, we get the following.

▶ **Lemma 8.** *If a program $\mathcal{P}$ is robust against* `CM`*, then $\mathcal{P}$ is robust against* `CCv`*.*

[2] Given two concurrent transactions t_1 and t_2 that write on a common variable x, $\mathsf{isu}(p_1, t_1)$ is in store order before $\mathsf{del}(p_1, t_2)$ and $\mathsf{isu}(p_2, t_2)$ before $\mathsf{del}(p_2, t_1)$.

5.3 Robustness Violations under Weak Causal Consistency

If a program is robust against CM, then it must not contain a write-write race under CM (note that this is not true for CCv). Therefore, by Theorem 2, a program which is robust against CM has the same set of traces under both CM and CC, which implies that it is also robust against CC. Conversely, since CC is weaker than CM (i.e., $\mathbb{T}\mathsf{r}_{\mathtt{CM}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\mathtt{CC}}(\mathcal{P})$ for any $\mathcal{P}$), if a program is robust against CC then it is robust against CM. Thus, we obtain the following.

▶ **Theorem 9.** *A program $\mathcal{P}$ is robust against* CC *iff it is robust against* CM.

6 Reduction to SC Reachability

We present a reduction of robustness checking to a reachability problem in a program executing under the serializability semantics. Given a program $\mathcal{P}$ and a semantics $\mathsf{X} \in \{\mathtt{CCv}, \mathtt{CM}, \mathtt{CC}\}$, we define an instrumentation of $\mathcal{P}$ such that $\mathcal{P}$ is not robust against X iff the instrumentation reaches an error state under serializability. The instrumentation uses auxiliary variables to simulate the robustness violations (in particular, the delayed transactions) satisfying the patterns given in Fig. 5. We focus our presentation on the second violation pattern of CCv (similar to the second violation pattern of CM): $\tau_{\mathtt{CCv2}} = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_2 \cdot \mathsf{del}(p_0,t) \cdot \gamma$. We describe the instrumentation only informally, the precise definition can be found in [11].

The process p that delayed the first transaction t is called the *Attacker*. The processes delaying transactions in $\beta_1 \cdot \mathsf{isu}(p_1,t_1)$ are called *Visibility Helpers*. Recall that all the delayed transactions must be causally after $\mathsf{isu}(p,t)$ and causally before $\mathsf{isu}(p_1,t_1)$. The processes that execute transactions in β_2 and contribute to the happens-before path between $\mathsf{isu}(p_1,t_1)$ and $\mathsf{del}(p_0,t)$ are called *Happens-Before Helpers*. A happens-before helper cannot be the attacker or a visibility helper since this would contradict causal delivery (a transaction of a happens-before helper is not delayed, so it is visible immediately to all processes, and it cannot follow a delayed transaction). γ contains the stores of the delayed transactions from $\mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1)$. It is important to notice that we may have $t = t_1$. In this case, $\beta_1 = \epsilon$ and the only delayed transaction is t. Also, all delayed transactions in β_1 including t_1 may be issued on the same process as t. In all these cases, the set of Visibility Helpers is empty.

The instrumentation uses two copies of the set of shared variables in the original program. We use primed variables x' to denote the second copy. When a process becomes the attacker or a visibility helper, it will write only to the second copy that is visible only to these processes (and remains invisible to the other processes including the happens-before helpers). The writes made by the other processes including the happens-before helpers are made visible to all processes, i.e., they are applied on both copies of the shared variables.

To establish the causality chains of delayed transactions performed by the attacker and the visibility helpers, we look whether a transaction can extend the causality chain started by the first delayed transaction from the attacker. In order for a transaction to "join" the causality chain, it has to satisfy one of the following conditions:

- the transaction is issued by a process that has already another transaction in the causality chain. Thus, we ensure the continuity of the causality chain through program order.
- the transaction is reading from a variable updated by a previous transaction in the causality chain. Hence, we ensure the continuity of the causality chain through write-read.

We introduce a flag for each shared variable to mark the fact that it was updated by a previous transaction in the causality chain. These flags are used by the instrumentation to establish whether a transaction "joins" a causality chain. Enforcing a happens-before path starting in the last delayed transaction, using transactions of the happens-before helpers, can be done in the similar way. Compared to causality chains, there are two more cases in which a transaction can extend a happens-before path:

- the transaction writes to a variable read by a previous transaction in the happens-before path. Hence, we ensure the continuity of the happens-before path through read-write.
- the transaction writes to a variable updated by a previous transaction in the happens-before path. We ensure the continuity of the happens-before path through write-write.

We extend the set of flags used for causality chains to record if a variable was read or written by a previous transaction in the happens-before path. Overall, the instrumentation uses a flag $x.event$ or $x'.event$ for each (copy of a) shared variable, that records the type of the last access (read or write) to that variable. Initially, these flags and other flags used by the instrumentation as explained below are initialized to null ($\perp$).

In general, whether a process is an attacker, visibility helper, or happens-before helper is not enforced syntactically by the instrumentation, and can vary from execution to execution. The role of a process in an execution is set *non-deterministically* during the execution using some additional process-local flags. Thus, during an execution, each process chooses to set to true at most one of the flags $p.a$, $p.vh$, and $p.hbh$, implying that the process becomes an attacker, visibility helper, or happens-before helper, respectively. At most one process can be an attacker, i.e., set $p.a$ to true. Before a process becomes an attacker or visibility helper it passes though a stage where it executes transactions in the usual way without delaying them.

A process can non-deterministically choose to delay a transaction at which point it sets a *global* flag $a_{\mathsf{tr_A}}$ to true. During the delayed transaction it chooses randomly a write instruction to a shared variable y and stores the name of this variable in the global variable $a_{\mathsf{st_A}}$. The values written during delayed transactions are stored in primed variables and visible only to the attacker and the visibility helpers. Each time the attacker writes to a variable z' (the copy of z from the original program) during a delayed transaction, it sets the flag $z'.event$ to st which will allow other processes that read the same variable to join the set of visibility helpers and start delaying their transactions. Once the attacker delays a transaction, it starts reading only from the primed variables (i.e., z').

When $a_{\mathsf{tr_A}}$ is set to true by the attacker, other processes continue the execution of their original instructions but, whenever they store a value they write it to both the shared variable z and the primed variable z' so it is visible to all processes. When a process chooses non deterministically to join the visibility helpers, it delays all writes (i.e., writes only to primed variables) and reads only from the primed variables.

In order for the attacker or a visibility helper to start the happens-before path, it has to either read or write a shared variable x that was not accessed by a delayed transaction (i.e., $x'.event = \perp$). In this case we set the global flag HB to true to mark the start of the happens-before path and the end of the causality chain, and set the flag $x.event$ to ld. When HB becomes true the attacker and the visibility helpers stop executing new transactions.

Also, when HB becomes true, the remaining processes can choose non-deterministically to join the set of happens-before helpers, i.e., continue the happens-before path created by the existing happens-before helpers, the attacker, or visibility helper. The happens-before helpers continue executing their instructions, until one of them reads from the variable y whose name was stored in $a_{\mathsf{st_A}}$. This establishes a happens-before path between the last delayed transaction and a "fictitious" store event corresponding to the first delayed transaction that could be executed just after this read of y. The execution doesn't have to contain this store event explicitly since it is always enabled. Therefore, at the end of every transaction, the instrumentation checks whether the transaction reads y. If this is the case, then the execution stops and goes to an error state to indicate that this is a robustness violation.

The role of a process in an execution is chosen non-deterministically at runtime. Therefore, the instrumentation $[\![\mathcal{P}]\!]$ of a program $\mathcal{P}$ is obtained by replacing each instruction $\langle linst\rangle$ with the concatenation of the instrumentations corresponding to the attacker, the visibility helpers, and the happens-before helpers, i.e., $[\![\langle linst\rangle]\!] ::= [\![\langle linst\rangle]\!]_{\mathsf{A}}\ [\![\langle linst\rangle]\!]_{\mathsf{VH}}\ [\![\langle linst\rangle]\!]_{\mathsf{HbH}}$. The following theorem states the correctness of the instrumentation.

▶ **Theorem 10.** *A program $\mathcal{P}$ is not robust against* `CCv` *iff $[\![\mathcal{P}]\!]$ reaches the error state.*

A similar instrumentation can be defined for the other variations of causal consistency, i.e., causal memory (`CM`) and weak causal consistency (`CC`).

The following result states the complexity of checking robustness for finite-state programs [3] against one of the three variations of causal consistency considered in this work (we use causal consistency as a generic name to refer to all of them). It is a direct consequence of Theorem 10 and of previous results concerning the reachability problem in concurrent programs running over SC, with a fixed [27] or parametric number of processes [40].

▶ **Corollary 11.** *Checking robustness of finite-state programs against causal consistency is PSPACE-complete when the number of processes is fixed and EXPSPACE-complete, otherwise.*

7 Related Work

Causal consistency is one of the oldest consistency models for distributed systems [30]. Formal definitions of several variants of causal consistency, suitable for different types of applications, have been introduced recently [19, 18, 38, 14]. The definitions in this paper are inspired from these works and coincide with those given in [14]. In that paper, the authors address the decidability and the complexity of verifying that an implementation of a storage system is causally consistent (i.e., all its computations, for every client, are causally consistent).

While our paper focuses on *trace-based* robustness, *state-based robustness* requires that a program is robust if the set of all its reachable states under the weak semantics is the same as its set of reachable states under the strong semantics. While state-robustness is the necessary and sufficient concept for preserving state-invariants, its verification, which amounts in computing the set of reachable states under the weak semantics, is in general a hard problem. The decidability and the complexity of this problem has been investigated in the context of relaxed memory models such as TSO and Power, and it has been shown that it is either decidable but highly complex (non-primitive recursive), or undecidable [8, 9]. As far as we know, the decidability and complexity of this problem has not been investigated for causal consistency. Automatic procedures for approximate reachability/invariant checking have been proposed using either abstractions or bounded analyses, e.g., [10, 5, 21, 1]. Proof methods have also been developed for verifying invariants in the context of weakly consistent models such as [29, 26, 36, 4]. These methods, however, do not provide decision procedures.

Decidability and complexity of trace-based robustness has been investigated for the TSO and Power memory models [15, 13, 22]. The work we present in this paper borrows the idea of using minimal violation characterizations for building an instrumentation allowing to obtain a reduction of the robustness checking problem to the reachability checking problem over SC. However, applying this approach to the case of causal consistency is not straightforward and requires different proof techniques. Dealing with causal consistency is far more tricky and difficult than dealing with TSO, and requires coming up with radically different arguments

[3] That is, programs where the number of variables and the data domain are bounded.

and proofs, for (1) characterizing in a finite manner the set of violations, (2) showing that this characterization is sound and complete, and (3) using effectively this characterization in the definition of the reduction to the reachability problem.

As far as we know, our work is the first one that establishes results on the decidability and complexity issues of the robustness problem in the context of causal consistency, and taking into account transactions. The existing work on the verification of robustness for distributed systems consider essentially trace-based concepts of robustness and provide either over- or under-approximate analyses for checking it. The static analyses proposed in [12, 16, 17, 20] are based on computing an abstraction of the set of computations, which is used in searching for robustness violations. These approaches may return false alarms due to the abstractions they consider. In particular, [12] shows that a trace under causal convergence is not admitted by the serializability semantics iff it contains a (transactional) happens-before cycle with a RW dependency, and another RW or WW dependency. This characterization alone is not sufficient to prove the reduction in Theorem 10, which requires a finer characterization of robustness violations. A sound (but not complete) bounded analysis for detecting robustness violation is proposed in [35]. Our approach is technically different, is precise, and provides a decision procedure for checking robustness when the program is finite-state.

References

1 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. Context-Bounded Analysis for POWER. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 56–74, 2017.

2 Atul Adya. *Weak consistency: A generalized theory and optimistic implementations for distributed transactions.* PhD thesis, MIT, 1999.

3 Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1):37–49, 1995.

4 Jade Alglave and Patrick Cousot. Ogre and Pythia: an invariance proof method for weak consistency models. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 3–18. ACM, 2017.

5 Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

6 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

7 Sérgio Almeida, João Leitão, and Luís E. T. Rodrigues. ChainReaction: a causal+ consistent datastore based on chain replication. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 85–98. ACM, 2013.

8 Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 7–18. ACM, 2010.

9 Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. What's Decidable about Weak Memory Models? In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 26–46. Springer, 2012.

10 Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Getting Rid of Store-Buffers in TSO Analysis. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2011.

11 Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Robustness Against Transactional Causal Consistency. *CoRR*, 2019.

12 Giovanni Bernardi and Alexey Gotsman. Robustness against Consistency Models with Atomic Visibility. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

13 Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and Enforcing Robustness against TSO. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 533–553. Springer, 2013.

14 Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 626–638. ACM, 2017.

15 Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding Robustness against Total Store Ordering. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 428–440. Springer, 2011.

16 Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 458–472. ACM, 2017.

17 Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. Static serializability analysis for causal consistency. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 90–104. ACM, 2018.

18 Sebastian Burckhardt. Principles of Eventual Consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150, 2014.

19 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 271–284. ACM, 2014.

20 Andrea Cerone and Alexey Gotsman. Analysing Snapshot Isolation. *J. ACM*, 65(2):11:1–11:41, 2018.

21 Andrei Marian Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. *Computer Languages, Systems & Structures*, 47:62–76, 2017.

22 Egor Derevenetc and Roland Meyer. Robustness against Power is PSpace-complete. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 158–170. Springer, 2014.

23 Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: scalable causal consistency using dependency matrices and physical clocks. In Guy M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 11:1–11:14. ACM, 2013.

24 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

25 Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

26 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 371–384. ACM, 2016.

27 Dexter Kozen. Lower Bounds for Natural Proof Systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 254–266. IEEE Computer Society, 1977.

28 Arthur Kurath. Analyzing Serializability of Cassandra Applications. Master's thesis, ETH Zurich, Switzerland, 2017.

29 Ori Lahav and Viktor Vafeiadis. Owicki-Gries Reasoning for Weak Memory Models. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2015.

30 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

31 Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

32 Richard J Lipton and Jonathan S Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton University, Department of Computer Science, August 1988.

33 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 401–416. ACM, 2011.

34 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 313–328. USENIX Association, 2013.

35 Kartik Nagar and Suresh Jagannathan. Automatic Detection of Serializability Violations Under Weak Consistency. In *29th Intern. Conf. on Concurrency Theory (CONCUR'18)*, September 2018. to appear.

36 Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE tool: proving weakly-consistent applications correct. In Peter Alvaro and Alysson Bessani, editors, *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, pages 2:1–2:3. ACM, 2016.

37 Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

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

39 Nuno M. Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. In *33rd IEEE International Symposium on Reliable Distributed Systems Workshops, SRDS Workshops 2014, Nara, Japan, October 6-9, 2014*, pages 30–33. IEEE Computer Society, 2014.

40 Charles Rackoff. The Covering and Boundedness Problems for Vector Addition Systems. *Theor. Comput. Sci.*, 6:223–231, 1978.

41 Dennis E. Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.