On Verifying Causal Consistency

Ahmed Bouajjani
Univ. Paris Diderot & IUF, France
abou@irif.fr

Constantin Enea
Univ. Paris Diderot, France
cenea@irif.fr

Rachid Guerraoui
EPFL, Switzerland
rachid.guerraoui@epfl.ch

Jad Hamza
EPFL-INRIA, Switzerland
jad.hamza@epfl.ch

Abstract

Causal consistency is one of the most adopted consistency criteria for distributed implementations of data structures. It ensures that operations are executed at all sites according to their causal precedence. We address the issue of verifying automatically whether the executions of an implementation of a data structure are causally consistent. We consider two problems: (1) checking whether *one* single execution is causally consistent, which is relevant for developing testing and bug finding algorithms, and (2) verifying whether *all* the executions of an implementation are causally consistent.

We show that the first problem is NP-complete. This holds even for the read-write memory abstraction, which is a building block of many modern distributed systems. Indeed, such systems often store data in key-value stores, which are instances of the read-write memory abstraction. Moreover, we prove that, surprisingly, the second problem is *undecidable*, and again this holds even for the read-write memory abstraction. However, we show that for the read-write memory abstraction, these negative results can be circumvented if the implementations are *data independent*, i.e., their behaviors do not depend on the data values that are written or read at each moment, which is a realistic assumption.

We prove that for data independent implementations, the problem of checking the correctness of a single execution w.r.t. the read-write memory abstraction is polynomial time. Furthermore, we show that for such implementations the set of non-causally consistent executions can be represented by means of a finite number of *register automata*. Using these machines as observers (in parallel with the implementation) allows to reduce polynomially the problem of checking causal consistency to a state reachability problem. This reduction holds regardless of the class of programs used for the implementation, of the number of read-write variables, and of the used data domain. It allows leveraging existing techniques for assertion/reachability checking to causal consistency verification. Moreover, for a significant class of implementations, we derive from this reduction the decidability of verifying causal consistency w.r.t. the read-write memory abstraction.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Model checking; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification; E.1 [*Data structures*]: Distributed data structures

Keywords distributed systems, causal consistency, model checking, static program analysis

1. Introduction

Causal consistency [30] (CC for short) is one of the oldest and most widely spread correctness criterion for distributed systems. For a distributed system composed of several sites connected through a network where each site executes some set of operations, if an operation o_1 affects another operation o_2 (o_2 causally depends on o_1), causal consistency ensures that all sites must execute these operations in that order. There exist many efficient implementations satisfying this criterion, e.g., [5, 14, 15, 26, 33], contrary to strong consistency (linearizability) which cannot be ensured in the presence of network partitions and while the system remains available [18, 20] (the sites answer to clients' requests without delay).

However, developing distributed implementations satisfying causal consistency poses many challenges: Implementations may involve a large number of sites communicating through unbounded[1] communications channels. Roughly speaking, causal consistency can be ensured if each operation (issued by some site) is broadcast to the other sites together with its whole "causal past" (the other operations that affect the one being broadcast). But this is not feasible in practice, and various optimizations have been proposed that involve for instance the use of vector clocks [17, 34]. Defining and implementing such optimizations is generally very delicate and error prone. Therefore, it is appealing to consider formal methods to help developers write correct implementations. At different stages of the development, both testing and verification techniques are needed either for detecting bugs or for establishing correctness w.r.t abstract specifications. We study in this paper two fundamental problems in this context: (1) checking whether one given execution of an implementation is causally consistent, a problem that is relevant for the design of testing algorithms, and (2) the problem of verifying whether all the executions of an implementation are causally consistent.

First, we prove that checking causal consistency for a single execution is NP-hard in general. We prove in fact that this problem is NP-complete for the read-write memory abstraction (RWM for

[1] Throughout the paper, *unbounded* means finite but arbitrarily large.

short), which is at the basis of many distributed data structures used in practice.

Moreover, we prove that the problem of verifying causal consistency of an implementation is undecidable in general. We prove this fact in two different ways. First, we prove that for regular specifications (i.e., definable using finite-state automata), this problem is undecidable even for finite-state implementations with two sites communicating through bounded-size channels. Furthermore, we prove that even for the particular case of the RWM specification, the problem is undecidable in general. (The proof in this case is technically more complex and requires the use of implementations with more than two sites.)

This undecidability result might be surprising, since it is known that linearizability (stronger than CC) [24] and eventual consistency (weaker than CC) [40] are decidable to verify in that same setting [3, 8, 22]. This result reveals an interesting aspect in the definition of causal consistency. Intuitively, two key properties of causal consistency are that (1) it requires that the order between operations issued by the same site to be preserved globally at all the sites, and that (2) it allows an operation o_1 which happened arbitrarily sooner than an operation o_2 to be executed after o_2 (if o_1 and o_2 are not causally related). Those are the essential ingredients that are used in the undecidability proofs (that are based on encodings of the Post Correspondence Problem). In comparison, linearizability does not satisfy (2) because for a fixed number of sites/threads, the reordering between operations is bounded (since only operations which overlap in time can be reordered), while eventual consistency does not satisfy (1).

Our NP-hardness and undecidability results show that reasoning about causal consistency is intrinsically hard in general. However, by focusing on the case of the RWM abstraction, and by considering commonly used objects that are instances of this abstraction, e.g., key-value stores, one can observe that their implementations are typically *data independent* [1, 43]. This means that the way these implementations handle data with read and write instructions is insensitive to the actual data values that are read or written. We prove that reasoning about causal consistency w.r.t. the RWM abstraction becomes tractable under the natural assumption of data independence. More precisely, we prove that checking causal consistency for a single computation is polynomial in this case, and that verifying causal consistency of an implementation is polynomially reducible to a state reachability problem, the latter being decidable for a significant class of implementations. Let us explain how we achieve that.

In fact, data independence implies that it is sufficient to consider executions where each value is written at most once; let us call such executions *differentiated* (see, e.g., [1]). The key step toward the results mentioned above is a characterization of the set of all differentiated executions that violate causal consistency w.r.t. the RWM. This characterization is based on the notion of a *bad pattern* that can be seen as a set of operations occurring (within an execution) in some particular order corresponding to a causal consistency violation. We express our bad patterns using appropriately defined conflict/dependency relations between operations along executions. We show that there is a *finite number* of bad patterns such that an execution is consistent w.r.t. the RWM abstraction *if and only if* the execution does not contain any of these patterns.

In this characterization, the fact that we consider only differentiated executions is crucial. The reason is that all relations used to express bad patterns include the read-from relation that associates with each read operation the write operation that provides its value. This relation is uniquely defined for differentiated executions, while for arbitrary executions where writes are not unique, reads can take their values from an arbitrarily large number of writes. This is actually the source of complexity and undecidability in the non-data independent case.

Then, we exploit this characterization in two ways. First, we show that for a given execution, checking that it contains a bad pattern can be done in polynomial time, which constitutes an important gain in complexity w.r.t. to the general algorithm that does not exploit data independence (precisely because the latter needs to consider all possible read-from relations in the given execution.)

Furthermore, we show that for each bad pattern, it is possible to construct effectively an *observer* (which is a state-machine of some kind) that is able, when running in parallel with an implementation, to detect all the executions containing the bad pattern. A crucial point is to show that these observers are in a class of state-machines that has “good” decision properties. (Basically, it is important that checking whether they detect a violation is decidable for a significant class of implementations.) We show that the observers corresponding to the bad patterns we identified can be defined as *register automata* [11], i.e., finite-state state machines supplied with a finite number of registers that store data over a potentially infinite domain (such as integers, strings, etc.) but on which the only allowed operation is checking equality. An important feature of these automata is that their state reachability problem can be reduced to the one for (plain) finite-state machines. The construction of the observers is actually independent from the type of programs used for the implementation, leading to a semantically sound and complete reduction to a state reachability problem (regardless of the decidability issue) even when the implementation is deployed over an unbounded number of sites, has an unbounded number of variables (keys) storing data over an unbounded domain.

Our reduction enables the use of any reachability analysis or assertion checking tool for the verification of causal consistency. Moreover, for an important class of implementations, this reduction leads to decidability and provides a verification algorithm for causal consistency w.r.t. the RWM abstraction. We consider implementations consisting of a finite number of state machines communicating through a network (by message passing). Each machine has a finite number of finite-domain (control) variables with unrestricted use, in addition to a finite number of *data variables* that are used only to store and move data, and on which no conditional tests can be applied. Moreover, we do not make any assumption on the network: the machines communicate through unbounded unordered channels, which is the usual setting in large-scale distributed networks. (Implementations can apply ordering protocols on top of this most permissive model.)

Implementations in the class we consider have an infinite number of configurations (global states) due to (1) the unboundedness of the data domain, and (2) the unboundedness of the communication channels. First, we show that due to data independence and the special form of the observers detecting bad patterns, proving causal consistency for any given implementation in this class (with any data domain) reduces to proving its causal consistency for a *bounded* data domain (with precisely 5 elements). This crucial fact allows to get rid of the first source of unboundedness in the configuration space. The second source of unboundedness is handled using counters: we prove that checking causal consistency in this case can be reduced to the state reachability problem in Vector Addition Systems with States (equivalent to unbounded Petri Nets), and conversely. This implies that verifying causal consistency w.r.t. the RWM (for this class of implementations) is EXPSPACE-complete.

It is important to notice that causal consistency has different meanings depending on the context and the targeted applications. Several efforts have been made recently for formalizing various notions of causal consistency (e.g., [12, 13, 21, 36, 38]). In this paper we consider three important variants. The variant called simply causal consistency (abbreviated as `CC`) allows non-causally dependent operations to be executed in different orders by different sites, and decisions about these orders to be revised by each site. This models mechanisms for solving the conflict between non-causally dependent operations where each site speculates on an order between such operations and possibly roll-backs some of them if needed later in the execution, e.g., [6, 28, 35, 40]. We also consider two stronger notions, namely *causal memory* (`CM`) [2, 36], and *causal convergence* (`CCv`) [12, 13, 36]. The latter assumes that there is a total order between non-causally dependent operations and each site can execute operations only in that order (when it sees them). Therefore, a site is not allowed to revise its ordering of non-causally dependent operations, and all sites execute in the same order the operations that are visible to them. This notion is used in a variety of systems [5, 15, 29, 39, 41, 44] because it also implies a strong variant of convergence, i.e., that every two sites that receive the same set of updates execute them in the same order. As for `CM`, a site is allowed to diverge from another site on the ordering of non-causally dependent operations, but is not allowed to revise its ordering later on. `CM` and `CCv` are actually incomparable [36].

All the contributions we have described above in this section hold for the `CC` criterion. In addition, concerning `CM` and `CCv`, we prove that (1) the NP-hardness and undecidability results hold, (2) a characterization by means of a finite number of bad patterns is possible, and (3) checking consistency for a single execution is polynomial time.

To summarize, this paper establishes the first complexity and (un)decidability results concerning the verification of causal consistency:

- NP-hardness of the problems of checking `CC`, `CM`, and `CCv` for a single execution (Section 5).
- Undecidability of the problems of verifying `CC`, `CM`, and `CCv` for regular specifications, and actually even for the RWM specification (Section 6).
- A polynomial-time procedure for verifying that a single execution of a data independent implementation is `CC`, `CM`, and `CCv` w.r.t. RWM (Section 8).
- Decidability and complexity for the verification of `CC` w.r.t. the RWM for a significant class of data independent implementations (Section 10).

The complexity and decidability results obtained for the RWM (under the assumption of data independence) are based on two key contributions that provide a deep insight on the problem of verifying causal consistency, and open the door to efficient automated testing/verification techniques:

- A characterization as a finite set of "bad patterns" of the set of violations to `CC`, `CM`, and `CCv` w.r.t. the RWM, under the assumption of data independence (Section 7).
- A polynomial reduction of the problem of verifying that a data independent implementation is `CC` w.r.t. to the RWM to a state reachability (or dually to an invariant checking) problem (Section 9).

Some proofs are deferred to the long version [10].

2. Notations

2.1 Sets, Multisets, Relations

Given a set O and a relation $\mathcal{R} \subseteq O \times O$, we denote by $o_1 <_{\mathcal{R}} o_2$ the fact that $(o_1, o_2) \in \mathcal{R}$. We denote by $o_1 \leq_{\mathcal{R}} o_2$ the fact that $o_1 <_{\mathcal{R}} o_2$ or $o_1 = o_2$. We denote by $\mathcal{R}^+$ the *transitive closure* of $\mathcal{R}$, which is the composition of one or more copies of $\mathcal{R}$.

Let O' be a subset of O. Then $\mathcal{R}_{|O'}$ is the relation $\mathcal{R}$ projected on the set O', that is $\{(o_1, o_2) \in \mathcal{R} \mid o_1, o_2 \in O'\}$. The set $O' \subseteq O$ is said to be *downward-closed* (with respect to relation $\mathcal{R}$) if $\forall o_1, o_2$, if $o_2 \in O'$ and $o_1 <_{\mathcal{R}} o_2$, then $o_1 \in O'$ as well.

$$\begin{tikzcd}
\texttt{CCv} \arrow[dr] & \\
 & \texttt{CC} \\
\texttt{CM} \arrow[ur] &
\end{tikzcd}$$

Figure 1: Implication graph of causal consistency definitions.

2.2 Labeled Posets

A relation $< \subseteq O \times O$ is a *strict partial order* if it is transitive and irreflexive. A *poset* is a pair $(O, <)$ where $<$ is a strict partial order over O. Note here that we use the strict version of posets, and not the ones where the underlying partial order is *weak*, i.e. reflexive, antisymmetric, and transitive.

Given a set Σ, a Σ *labeled poset* ρ is a tuple $(O, <, \ell)$ where $(O, <)$ is a poset and $\ell : O \to \Sigma$ is the labeling function.

We say that ρ' is a *prefix* of ρ if there exists a downward closed set $A \subseteq O$ (with respect to relation $<$) such that $\rho' = (A, <, \ell)$. A (resp., labeled) *sequential poset* (sequence for short) is a (resp., labeled) poset where the relation $<$ is a strict total order. We denote by $e \cdot e'$ the concatenation of sequential posets.

3. Replicated Objects

We define an abstract model for the class of distributed objects called *replicated objects* [7], where the object state is replicated at different sites in a network, called also *processes*, and updates or queries to the object can be submitted to any of these sites. This model reflects the view that a client has on an execution of this object, i.e., a set of operations with their inputs and outputs where every two operations submitted to the same site are ordered. It abstracts away the implementation internals like the messages exchanged by the sites in order to coordinate about the object state. Such a partially ordered set of operations is called a *history*. The correctness (consistency) of a replicated object is defined with respect to a *specification* that captures the behaviors of that object in the context of sequential programs.

3.1 Histories

A replicated object implements a programming interface (API) defined by a set of *methods* $\mathbb{M}$ with input or output values from a domain $\mathbb{D}$.

For instance, in the case of the read/write memory, the set of methods $\mathbb{M}$ is $\{\mathtt{wr}, \mathtt{rd}\}$ for writing or reading a variable. Also, given a set of *variables* $\mathbb{X}$, the domain $\mathbb{D}$ is defined as $(\mathbb{X} \times \mathbb{N}) \uplus \mathbb{X} \uplus \mathbb{N} \uplus \{\perp\}$. Write operations take as input a variable in $\mathbb{X}$ and a value in $\mathbb{N}$ and return $\perp$ while read operations take as input a variable in $\mathbb{X}$ and return a value in $\mathbb{N}$. The return value $\perp$ is often omitted for better readability.

A *history* $h = (O, \mathsf{PO}, \ell)$ is a poset labeled by $\mathbb{M} \times \mathbb{D} \times \mathbb{D}$, where:

- O is a set of operation identifiers, or simply *operations*,
- PO is a union of total orders between operations called *program order*: for $o_1, o_2 \in O$, $o_1 <_{\mathsf{PO}} o_2$ means that o_1 and o_2 were submitted to the same site, and o_1 occurred before o_2,
- for $m \in \mathbb{M}$ and $arg, rv \in \mathbb{D}$, and $o \in O$, $\ell(o) = (m, arg, rv)$ means that operation o is an invocation of m with input arg and returning rv. The label $\ell(o)$ is sometimes denoted $m(arg) \rhd rv$.

Given an operation o from a read/write memory history, whose label is either $\mathtt{wr}(x, v)$ or $\mathtt{rd}(x) \rhd v$, for some $x \in \mathbb{X}$, $v \in \mathbb{D}$, we define $\mathsf{var}(o) = x$ and $\mathsf{value}(o) = v$.

3.2 Specification

The consistency of a replicated object is defined with respect to a particular specification, describing the correct behaviors of that object in a sequential setting. A *specification* S is thus defined[2] as a set of sequences labeled by $\mathbb{M} \times \mathbb{D} \times \mathbb{D}$.

In this paper, we focus on the read/write memory whose specification S_{RW} is defined inductively as the smallest set of sequences closed under the following rules ($x \in \mathbb{X}$ and $v \in \mathbb{N}$):

1. $\varepsilon \in S_{\mathsf{RW}}$,
2. if $\rho \in S_{\mathsf{RW}}$, then $\rho \cdot \mathtt{wr}(x, v) \in S_{\mathsf{RW}}$,
3. if $\rho \in S_{\mathsf{RW}}$ contains no write on x, then $\rho \cdot \mathtt{rd}(x) \rhd 0 \in S_{\mathsf{RW}}$,
4. if $\rho \in S_{\mathsf{RW}}$ and the last write in ρ on variable x is $\mathtt{wr}(x, v)$, then $\rho \cdot \mathtt{rd}(x) \rhd v \in S_{\mathsf{RW}}$.

4. Causal Consistency

Causal consistency is one of the most widely used consistency criterion for replicated objects. Informally speaking, it ensures that, if an operation o_1 is *causally related* to an operation o_2 (e.g., some site knew about o_1 when executing o_2), then all sites must execute operation o_1 before operation o_2. Operations which are not causally related may be executed in different orders by different sites.

From a formal point of view, there are several variations of causal consistency that apply to slightly different classes of implementations. In this paper, we consider three such variations that we call causal consistency (CC), causal memory (CM), and causal convergence (CCv). We start by presenting CC followed by CM and CCv, which are both strictly stronger than CC. CM and CCv are not comparable (see Figure 1).

4.1 Causal Consistency: Informal Description

Causal consistency [21, 36] (CC for short) corresponds to the weakest notion of causal consistency that exists in the literature. We describe the intuition behind this notion of consistency using several examples, and then give the formal definition.

Recall that a history h models the point of view of a client using a replicated object, and it contains no information regarding the internals of the implementation, in particular, the messages exchanged between sites. This means that a history contains no notion of *causality order*. Thus, from the point of view of the client, a history is CC as long as there *exists* a causality order which explains the return value of each operation. This is why, in the formal definition of CC given in the next section, the causality order co is existentially quantified.

Example 1. *History (2e) is not* CC*. The reason is that there does not exist a causality order which explains the return values of all operations in the history. Intuitively, in any causality order,* $\mathtt{wr}(y, 1)$ *must be causally related to* $\mathtt{rd}(y) \rhd 1$ *(so that the read can return value* 1*). By transitivity of the causality order and because any causality order must contain the program order,* $\mathtt{wr}(x, 1)$ *must be causally related to* $\mathtt{wr}(x, 2)$*. However, site* p_c *first reads* $\mathtt{rd}(x) \rhd 2$*, and then* $\mathtt{rd}(x) \rhd 1$*. This contradicts the informal constraint that every site must see operations which are causally related in the same order.*

Example 2. *History (2c) is* CC*. The reason is that we can define a causality order where the writes* $\mathtt{wr}(x, 1)$ *and* $\mathtt{wr}(x, 2)$ *are not causally related between them, but each write is causally related to both reads. Since the writes are not causally related, site* p_b *can read them in any order.*

There is a subtlety here. In History (2c), site p_b first does $\mathtt{rd}(x) \rhd 1$, which implicitly means that it executed $\mathtt{wr}(x, 1)$ after $\mathtt{wr}(x, 2)$. Then p_b does $\mathtt{rd}(x) \rhd 2$ which means that p_b "changed its mind", and decided to order $\mathtt{wr}(x, 2)$ after $\mathtt{wr}(x, 1)$. This is allowed by CC, but as we will see later, not allowed by the stronger criteria CM and CCv.

[2] In general, specifications can be defined as sets of posets instead of sequences. This is to model conflict-resolution policies which are more general than choosing a total order between operations. In this paper, we focus on the read/write memory whose specification is a set of sequences.

p_a:	p_b:
$\mathtt{wr}(x, 1)$	$\mathtt{wr}(x, 2)$
$\mathtt{rd}(x) \rhd 2$	$\mathtt{rd}(x) \rhd 1$

(a) CM but not CCv

p_a:	p_b:
$\mathtt{wr}(z, 1)$	$\mathtt{wr}(x, 2)$
$\mathtt{wr}(x, 1)$	$\mathtt{rd}(z) \rhd 0$
$\mathtt{wr}(y, 1)$	$\mathtt{rd}(y) \rhd 1$
	$\mathtt{rd}(x) \rhd 2$

(b) CCv but not CM

p_a:	p_b:
$\mathtt{wr}(x, 1)$	$\mathtt{wr}(x, 2)$
	$\mathtt{rd}(x) \rhd 1$
	$\mathtt{rd}(x) \rhd 2$

(c) CC but not CM nor CCv

p_a:	p_b:
$\mathtt{wr}(x, 1)$	$\mathtt{wr}(x, 2)$
$\mathtt{rd}(y) \rhd 0$	$\mathtt{rd}(y) \rhd 0$
$\mathtt{wr}(y, 1)$	$\mathtt{wr}(y, 2)$
$\mathtt{rd}(x) \rhd 1$	$\mathtt{rd}(x) \rhd 2$

(d) CC, CM and CCv but not sequentially consistent

p_a:	p_b:	p_c:
$\mathtt{wr}(x, 1)$	$\mathtt{rd}(y) \rhd 1$	$\mathtt{rd}(x) \rhd 2$
$\mathtt{wr}(y, 1)$	$\mathtt{wr}(x, 2)$	$\mathtt{rd}(x) \rhd 1$

(e) not CC (nor CM, nor CCv)

Figure 2: Histories showing the differences between the consistency criteria CC, CM, and CCv.

AxCausal	$\mathsf{PO} \subseteq co$
AxArb	$co \subseteq arb$
AxCausalValue	$\mathsf{CausalHist}(o)\{o\} \le \rho_o$
AxCausalSeq	$\mathsf{CausalHist}(o)\{\mathsf{POPast}(o)\} \le \rho_o$
AxCausalArb	$\mathsf{CausalArb}(o)\{o\} \le \rho_o$

where:
$\mathsf{CausalHist}(o) = (\mathsf{CausalPast}(o), co, \ell)$
$\mathsf{CausalArb}(o) = (\mathsf{CausalPast}(o), arb, \ell)$
$\mathsf{CausalPast}(o) = \{o' \in O \mid o' \le_{co} o\}$
$\mathsf{POPast}(o) = \{o' \in O \mid o' \le_{\mathsf{PO}} o\}$

Table 1: Axioms used in the definitions of causal consistency.

This feature of CC is useful for systems which do speculative executions and rollbacks [6, 40]. It allows systems to execute operations by speculating on an order, and then possibly rollback, and change the order of previously executed operations. This happens in particular in systems where convergence is important, where a consensus protocol is running in the background to make all sites eventually agree on a total order of operations. The stronger definitions, CM and CCv, are not suited to represent such speculative implementations.

4.2 Causal Consistency: Definition

We now give the formal definition of CC, which corresponds to the description of the previous section. A history $h = (O, \mathsf{PO}, \ell)$ is CC with respect to a specification S when there exists a strict partial order $co \subseteq O \times O$, called *causal order*, such that, for all operations $o \in O$, there exists a specification sequence $\rho_o \in S$ such that axioms AxCausal and AxCausalValue hold (see Table 1).

Axiom AxCausal states that the causal order must at least contain the program order. Axiom AxCausalValue states that, for each operation $o \in O$, the causal history of o (roughly, all the operations

ρ_a:

$$\begin{tikzcd}[o_1]\,\mathtt{wr}(x,1) \arrow[dr] & \\ & [o_3]\,\mathtt{rd}(x) \\ [o_2]\,\mathtt{wr}(y,2) \arrow[ur] & \end{tikzcd}$$

ρ_b:

$$\begin{tikzcd}[o_1]\,\mathtt{wr}(x,1) \arrow[d] \\ [o_2]\,\mathtt{wr}(y,2) \arrow[d] \\ [o_3]\,\mathtt{rd}(x) \rhd 1\end{tikzcd}$$

ρ_c:

$$\begin{tikzcd}[o_1]\,\mathtt{wr}(x,1) \\ [o_2]\,\mathtt{wr}(y,2) \\ [o_3]\,\mathtt{rd}(x) \rhd 1\end{tikzcd}$$

Figure 3: Illustration of the $\leq$ relation. We have $\rho_a \leq \rho_b$, but not $\rho_a \leq \rho_c$. The label of an operation o is written next to o. The arrows represent the transitive reduction of the strict partial orders underlying the labeled posets. (For instance, none of the operations in ρ_c are ordered.)

which are before o in the causal order) can be sequentialized in order to obtain a valid sequence of the specification S. This sequentialization must also preserve the constraints given by the causal order. Formally, we define the *causal past* of o, $\mathsf{CausalPast}(o)$, as the *set* of operations before o in the causal order and the *causal history* of o, $\mathsf{CausalHist}(o)$ as the restriction of the causal order to the operations in its causal past. Since a site is not required to be consistent with the return values it has provided in the past or the return values provided by the other sites, the axiom $\mathsf{AxCausalValue}$ uses the causal history where only the return value of operation o has been kept. This is denoted by $\mathsf{CausalHist}(o)\{o\}$. The fact that the latter can be sequentialized to a sequence ρ_o in the specification is denoted by $\mathsf{CausalHist}(o)\{o\} \leq \rho_o$. We defer the formal definition of these two last notations to the next section.

4.3 Operations on Labeled Posets

First, we introduce an operator which projects away the return values of a subset of operations. Let $\rho = (O, <, \ell)$ be a $\mathbb{M} \times \mathbb{D} \times \mathbb{D}$ labeled poset and $O' \subseteq O$. We denote by $\rho\{O'\}$ the labeled poset where only the return values of the operations in O' have been kept. Formally, $\rho\{O'\}$ is the $(\mathbb{M} \times \mathbb{D}) \cup (\mathbb{M} \times \mathbb{D} \times \mathbb{D})$ labeled poset $(O, <, \ell')$ where for all $o \in O'$, $\ell'(o) = \ell(o)$, and for all $o \in O \setminus O'$, if $\ell(o) = (m, arg, rv)$, then $\ell'(o) = (m, arg)$. If $O' = \{o\}$, we denote $\rho\{O'\}$ by $\rho\{o\}$.

Second, we introduce a relation on labeled posets, denoted $\leq$. Let $\rho = (O, <, \ell)$ and $\rho' = (O, <', \ell')$ be two posets labeled by $(\mathbb{M} \times \mathbb{D}) \cup (\mathbb{M} \times \mathbb{D} \times \mathbb{D})$ (the return values of some operations in O might not be specified). We denote by $\rho' \leq \rho$ the fact that ρ' has less order and label constraints on the set O. Formally, $\rho' \leq \rho$ if $<' \subseteq <$ and for all operation $o \in O$, and for all $m \in \mathbb{M}$, $arg, rv \in \mathbb{D}$,

- $\ell(o) = \ell'(o)$, or
- $\ell(o) = (m, arg, rv)$ implies $\ell'(o) = (m, arg)$.

Example 3. *For any set of operations $O' \subseteq O$, $\rho\{O'\} \leq \rho$. The reason is that $\rho\{O'\}$ has the same order constraints on O than ρ, but some return values are hidden in $\rho\{O'\}$.*

Example 4. *In Figure 3, we have $\rho_a \leq \rho_b$, as the only differences between ρ_a and ρ_b is the label of o_3, and the fact that $o_1 < o_2$ holds in ρ_b but not in ρ_a.*

We have $\rho_a \nleq \rho_c$, as $o_1 < o_3$ holds in ρ_a, but not in ρ_c.

4.4 Causal Memory (CM)

Compared to causal consistency, causal memory [2, 36] (denoted CM) does not allow a site to "change its mind" about the order of operations. The original definition of causal memory of Ahamad et al. [2] applies only to the $\mathsf{read/write\ memory}$ and it was extended by Perrin et al. [36] to arbitrary specifications. We use the more general definition, since it was also shown that it coincides with the original one for histories where for each variable $x \in \mathbb{X}$, the values written to x are unique.

For instance, History (2c) is CC but not CM. Intuitively, the reason is that site p_b first decides to order $\mathtt{wr}(x,1)$ after $\mathtt{wr}(x,2)$ (for $\mathtt{rd}(x) \rhd 1$) and then decides to order $\mathtt{wr}(x,2)$ after $\mathtt{wr}(x,1)$ (for $\mathtt{rd}(x) \rhd 2$).

On the other hand, History (2a) is CM. Sites p_a and p_b disagree on the order of the two write operations, but this is allowed by CM, as we can define a causality order where the two writes are not causally related.

Formally, a history $h = (O, \mathsf{PO}, \ell)$ is CM with respect to a specification S if there exists a strict partial order $co \subseteq O \times O$ such that, for each operation $o \in O$, there exists a specification sequence $\rho_o \in S$ such that axioms $\mathsf{AxCausal}$ and $\mathsf{AxCausalSeq}$ hold. With respect to CC, causal memory requires that each site is consistent with respect to the return values it has provided in the past. A site is still not required to be consistent with the return values provided by other sites. Therefore, $\mathsf{AxCausalSeq}$ states:

$$\mathsf{CausalHist}(o)\{\mathsf{POPast}(o)\} \leq \rho_o$$

where $\mathsf{CausalHist}(o)\{\mathsf{POPast}(o)\}$ is the causal history where only the return values of the operations which are before o in the program order (in $\mathsf{POPast}(o)$) are kept. For finite histories, if we set o to be the last operation of a site p, this means that we must explain all return values of operations in p by a single sequence $\rho_o \in S$. In particular, this is not possible for for site p_b in History (2c).

The following lemma gives the relationship between CM and CC.

Lemma 1 ([36]). *If a history h is* CM *with respect to a specification S, then h is* CC *with respect to S.*

Proof. We know by definition that there exists a strict partial order co such that, for all operation $o \in O$, there exists $\rho_o \in S$ such that axioms $\mathsf{AxCausal}$ and $\mathsf{AxCausalSeq}$. In particular, for any $o \in O$, we have $\mathsf{CausalHist}(o)\{\mathsf{POPast}(o)\} \leq \rho_o$.

Since $\mathsf{CausalHist}(o)\{o\} \leq \mathsf{CausalHist}(o)\{\mathsf{POPast}(o)\}$, and the relation $\leq$ is transitive, we have $\mathsf{CausalHist}(o)\{o\} \leq \rho_o$, and axiom $\mathsf{AxCausalValue}$ holds. $\square$

4.5 Causal Convergence (CCv)

Our formalization of causal convergence (denoted CCv) corresponds to the definition of *causal consistency* given in Burckhardt et al. [13] and Burckhardt [12] restricted to sequential specifications. CCv was introduced in the context of *eventual consistency*, another consistency criterion guaranteeing that roughly, all sites eventually converge towards the same state, when no new updates are submitted.

Causal convergence uses a total order between all the operations in a history, called the *arbitration order*, as an abstraction of the conflict resolution policy applied by sites to agree on how to order operations which are *not* causally related. As it was the case for the causal order, the arbitration order, denoted by arb, is not encoded explicitly in the notion of history and it is existentially quantified in the definition of CCv.

Example 5. *History (2a) is not* CCv*. The reason is that, for the first site p_a to read $\mathtt{rd}(x) \rhd 2$, the write $\mathtt{wr}(x,2)$ must be after $\mathtt{wr}(x,1)$ in the arbitration order. Symmetrically, because of the $\mathtt{rd}(x) \rhd 1$, $\mathtt{wr}(x,2)$ must be before $\mathtt{wr}(x,1)$ in the arbitration order, which is not possible.*

Example 6. *History (2b) gives a history which is* CCv *but not* CM*. To prove that it is* CCv*, a possible arbitration order is to have the writes of p_a all before the $\mathtt{wr}(x,2)$ operation, and the causality order then relates $\mathtt{wr}(y,1)$ to $\mathtt{rd}(y) \rhd 1$.*

On the other hand, History (2b) is not CM*. If History (2b) were* CM*, for site p_b, $\mathtt{wr}(y,1)$ should go before $\mathtt{rd}(y) \rhd 1$. By transitivity, this implies that $\mathtt{wr}(x,1)$ should go before $\mathtt{rd}(x) \rhd 2$. But for the* *read* $\mathtt{rd}(x) \rhd 2$ *to return value* 2*,* $\mathtt{wr}(x, 1)$ *should then also go before* $\mathtt{wr}(x, 2)$*. This implies that* $\mathtt{wr}(z, 1)$ *goes before* $\mathtt{rd}(z) \rhd 0$ *preventing* $\mathtt{rd}(z) \rhd 0$ *from reading the initial value* 0*.*

Example 7. *History (2d) shows that all causal consistency definitions (*CC*,* CM*, and* CCv*) are strictly weaker than sequential consistency. Sequential consistency [31] imposes a total order on all (read and write) operations. In particular, no such total order can exist for History (2d). Because of the initial writes* $\mathtt{wr}(x, 1)$ *and* $\mathtt{wr}(x, 2)$*, and the final reads* $\mathtt{rd}(x) \rhd 1$ *and* $\mathtt{rd}(x) \rhd 2$*, all the operations of* p_a *must be completely ordered before the operations of* p_b*, or vice versa. This would make one of the* $\mathtt{rd}(y) \rhd 0$ *to be ordered after either* $\mathtt{wr}(y, 1)$ *or* $\mathtt{wr}(y, 2)$*, which is not allowed by the* read/write memory *specification. On the other hand, History (2d) satisfies all criteria* CC*,* CM*,* CCv*. The reason is that we can set the causality order to not relate any operation from* p_a *to* p_b *nor from* p_b *to* p_a*.*

Formally, a history h is CCv with respect to S if there exist a strict partial order $co \subseteq O \times O$ and a strict total order $arb \subseteq O \times O$ such that, for each operation $o \in O$, there exists a specification sequence $\rho_o \in S$ such that the axioms AxCausal, AxArb, and AxCausalArb hold. Axiom AxArb states that the arbitration order arb must at least respect the causal order co. Axiom AxCausalArb states that, to explain the return value of an operation o, we must sequentialize the operations which are in the causal past of o, while respecting the arbitration order arb.

Axioms AxCausalArb and AxArb imply axiom AxCausalValue, as the arbitration order arb contains the causality order co. We therefore have the following lemma.

Lemma 2 ([36])**.** *If a history* h *is* CCv *with respect to a specification* S*, then* h *is* CC *with respect to* S*.*

Proof. Similar to the proof of Lemma 1, but using the fact that $\mathsf{CausalHist}(o)\{o\} \leq \mathsf{CausalArb}(o)\{o\}$ (since by axiom AxArb, $co \subseteq arb$). □

5. Single History Consistency is NP-complete

We first focus on the problem of checking whether a given history is consistent, which is relevant for instance in the context of testing a given replicated object. We prove that this problem is NP-complete for all the three variations of causal consistency (CC, CM, CCv) and the read/write memory specification.

Lemma 3. *Checking whether a history* h *is* CC *(resp.,* CM*, resp.,* CCv*) with respect to* S_{RW} *is* NP*-complete.*

Proof. Membership in NP holds for all the variations of causal consistency, and any specification S for which there is a polynomial-time algorithm that can check whether a given sequence is in S. This includes the read/write memory, and common objects such as sets, multisets, stacks, or queues. It follows from the fact that one can guess a causality order co (and an arbitration order arb for CCv), and a sequence ρ_o for each operation o, and then check in polynomial time whether the axioms of Table 1 hold, and whether $\rho_o \in S$.

For NP-hardness, we reduce boolean satisfiability to checking consistency of a single history reusing the encoding from Furbach et al. [19]. Let ϕ be a boolean formula in CNF with variables $x_1, \ldots, x_n$, and clauses $C_1, \ldots, C_k$. The goal is to define a history h which is CC if and only if ϕ is satisfiable. All operations on h are on a single variable y. For the encoding, we assume that each clause corresponds to a unique integer strictly larger than n.

For $i \in \{1, \ldots, n\}$, we define $Pos(x_i)$ as the set of clauses where x_i appears positively, and $Neg(x_i)$ as the set of clauses where x_i appears negatively.

For each $i \in \{1, \ldots, n\}$, h contains two sites, p^i_{false} and p^i_{true}. Site p^i_{false} first writes each $C \in Pos(x_i)$ (in the order they appear in $C_1, \ldots, C_k$) to variable y, and then, it writes i. Similarly, Site p^i_{true} writes each $C \in Neg(x_i)$ (in the order they appear in $C_1, \ldots, C_k$) to variable y, and then, it writes i.

Finally, a site p_{eval} does $\mathtt{rd}(y) \rhd 1 \cdots \mathtt{rd}(y) \rhd n$ followed by $\mathtt{rd}(y) \rhd C_1 \cdots \mathtt{rd}(y) \rhd C_k$.

We then prove the following equivalence: (the equivalence for CM and CCv can be proven similarly): h is CC iff ϕ is satisfiable.

($\Leftarrow$) This direction follows from the proof of [19]. They show that if ϕ is satisfiable, the history h is sequentially consistent.

($\Rightarrow$) Assume h is CC. Then, there exists co, such that, for all $o \in O$, there exists $\rho_o \in S_{\mathsf{RW}}$, such that AxCausal and AxCausalValue hold. In particular, each $\mathtt{rd}(y) \rhd i$ of p_{eval} must have a $\mathtt{wr}(y, i)$ in its causal past.

The $\mathtt{wr}(y, i)$ operation can either be from p^i_{false} (corresponding to setting variable x_i to false in ϕ), or from p^i_{true} (corresponding to setting variable x_i to true in ϕ). For instance, if it is from site p^i_{false}, then none of of the $\mathtt{wr}(y, C)$ for $C \in Pos(x_i)$ can be used for the reads $\mathtt{rd}(y) \rhd C_i$ of p_{eval}.

Consequently, for any variable x_i, only the writes of $\mathtt{wr}(y, C)$ for $C \in Pos(x_i)$, or the ones with $C \in Neg(x_i)$ can be used for the reads $\mathtt{rd}(y) \rhd C_i$ of p_{eval}.

Moreover, each read $\mathtt{rd}(y) \rhd C_i$ has a corresponding $\mathtt{wr}(y, C_i)$, meaning that ϕ is satisfiable. □

The reduction from boolean satisfiability used to prove NP-hardness uses histories where the same value is written multiple times on the same variable. We show in Section 8 that this is in fact necessary to obtain the NP-hardness: when every value is written only once per variable, the problem becomes polynomial time.

6. Undecidability of Verifying Causal Consistency

We now consider the problem of checking whether all histories of an implementation are causally consistent. We consider this problem for all variants of causal consistency (CC, CM, CCv).

We prove that this problem is undecidable. In order to formally prove the undecidability, we describe an abstract model for representing implementations.

6.1 Executions and Implementations

Concretely, an implementation is represented by a set of *executions*. Formally, an *execution* is a sequence of operations. Each operation is labeled by an element (p, m, arg, rv) of $\mathsf{PId} \times \mathbb{M} \times \mathbb{D} \times \mathbb{D}$, meaning that m was called with argument value arg on site p, and returned value rv. An *implementation* $\mathcal{I}$ is a set of executions which is prefix-closed (if $\mathcal{I}$ contains an execution $e \cdot e'$, $\mathcal{I}$ also contains e).

All definitions given for histories (and sets of histories) transfer to executions (and sets of executions) as for each execution e, we can define a corresponding history h. The history $h = (O, \mathsf{PO}, \ell)$ contains the same operations as e, and orders $o_1 <_{\mathsf{PO}} o_2$ if o_1 and o_2 are labeled by the same site, and o_1 occurs before o_2 in e.

For instance, an implementation is *data independent* if the corresponding set of histories is data independent.

6.2 Undecidability Proofs

We prove undecidability even when $\mathcal{I}$ and S are regular languages (given by regular expressions or by finite automaton). We refer to this as the first undecidability proof. Even stronger, we give a second undecidability proof, which shows that this problem is undecidable when the specification is set to S_{RW}, with a fixed number of variables, and with a fixed domain size (which is a particular regular language).

These results imply that the undecidability does not come from the expressiveness of the model used to describe implementations, nor from the complexity of the specification, but specifically from the fact that we are checking causal consistency.

For both undecidability proofs, our approach is to reduce the Post Correspondence Problem (PCP, an undecidable problem in formal languages), to the problem of checking whether $\mathcal{I}$ is *not* causally consistent (resp., CC,CM,CCv).

Definition 1. *Let Σ_{PCP} be a finite alphabet.* PCP *asks, given n pairs $(u_1, v_1), \dots, (u_n, v_n) \in (\Sigma_{\mathsf{PCP}}^* \times \Sigma_{\mathsf{PCP}}^*)$, whether there exist $i_1, \dots, i_k \in \{1, \dots, n\}$ such that $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$, with $(k > 0)$.*

From a high-level view, both proofs operate similarly. We build, from a PCP instance P, an implementation $\mathcal{I}$ (which is here a regular language) – and for the first proof, a specification S – such that P has a positive answer if and only if $\mathcal{I}$ contains an execution which is not causally consistent (resp., CC,CM,CCv) with respect to S (with respect to a bounded version of S_{RW} for the second proof).

The constructed implementations $\mathcal{I}$ produce, for each possible pair of words (u, v), an execution whose history $H_{(u,v)}$ is *not* causally consistent (resp., CC,CM,CCv) if and only if (u, v) form a *valid answer* for P.

Definition 2. *Two sequences (u, v) in Σ_{PCP}^* form a* valid answer *if $u = v$ and they can be decomposed into $u = u_{i_1} \cdot u_{i_2} \cdots u_{i_k}$ and $v = v_{i_1} \cdot v_{i_2} \cdots v_{i_k}$, with each (u_{i_j}, v_{i_j}) corresponding to a pair of problem P.*

Therefore, $\mathcal{I}$ is not causally consistent, if and only if $\mathcal{I}$ contains an execution whose history $H_{(u,v)}$ is not causally consistent, if and only if there exists (u, v) which form a valid answer for P, if and only if P has a positive answer.

6.3 Undecidability For Regular Specifications

For the first proof, we first prove that the *shuffling problem*, a problem on formal languages that we introduce, is not decidable. This is done by reducing PCP to the shuffling problem.

We then reduce the shuffling problem to checking whether an implementation is not causally consistent (resp., CC,CM,CCv), showing that verification of causal consistency is undecidable as well.

Given two words $u, v \in \Sigma^*$, the *shuffling* operator returns the set of words which can be obtained from u and v by interleaving their letters. Formally, we define $u \| v \subseteq \Sigma^*$ inductively: $\varepsilon \| v = \{v\}$, $u \| \varepsilon = \{u\}$ and $(a \cdot u) \| (b \cdot v) = a \cdot (u \| (b \cdot v)) \cup b \cdot ((a \cdot u) \| v)$, with $a, b \in \Sigma$.

Definition 3. *The* shuffling problem *asks, given a regular language L over an alphabet $(\Sigma_u \uplus \Sigma_v)^*$, if there exist $u \in \Sigma_u^*$ and $v \in \Sigma_v^*$ such that $u \| v \cap L = \varnothing$.*

Lemma 4. *The shuffling problem is undecidable.*

We now give the undecidability theorem for causal consistency (resp., CC,CM,CCv), by reducing the shuffling problem to the problem of verifying (non-)causal consistency. The idea is to let one site simulate words from Σ_u^*, and the second site from Σ_v^*. We then set the specification to be (roughly) the language L. We therefore obtain that there exists an execution which is *not* causally consistent with respect to L if and only if there exist $u \in \Sigma_u^*$, $v \in \Sigma_v^*$ such that no interleaving of u and v belongs to L, i.e. $u \| v \cap L = \varnothing$.

Theorem 1. *Given an implementation $\mathcal{I}$ and a specification S given as regular languages, checking whether all executions of $\mathcal{I}$ are causally consistent (resp.,* CC, CM, CCv*) with respect to S is undecidable.*

6.4 Undecidability for the Read/Write Memory Abstraction

Our approach for the second undecidability proof is to reduce directly PCP to the problem of checking whether a finite-state implementation is *not* CC (resp., CM, CCv) with respect to the read/write memory, without going through the shuffling problem. The reduction here is much more technical, and requires 13 sites. This is due to the fact that we cannot encode the constraints we want in the specification (as the specification is set to be S_{RW}), and we must encode them using appropriately placed read and write operations.

Theorem 2. *Given an implementation $\mathcal{I}$ as a regular language, checking whether all executions of $\mathcal{I}$ are causally consistent (resp.,* CC, CM, CCv*) with respect to S_{RW} is undecidable.*

7. Causal Consistency under Data Independence

Implementations used in practice are typically *data independent* [1], i.e. their behaviors do not depend on the particular data values which are stored at a particular variable. Under this assumption, we prove in Section 7.1 that it is enough to verify causal consistency for histories which use distinct wr values, called *differentiated* histories.

We then show in Section 7.2, for each definition of causal consistency, how to characterize non-causally consistent (differentiated) histories through the presence of certain sets of operations.

We call these sets of operations *bad patterns*, because any history containing one bad pattern is necessarily not consistent (for the considered consistency criterion). The bad patterns are defined through various relations derived from a differentiated history, and are all computable in polynomial time (proven in Section 8). For instance, for CC, we provide in Section 7.2 four bad patterns such that, a differentiated history h is CC if and only if h contains none of these bad patterns. We give similar lemmas for CM and CCv.

7.1 Differentiated Histories

Formally, a history (O, PO, ℓ) is said to be *differentiated* if for all $o_1 \neq o_2$, if $\ell(o_1) = \mathtt{wr}(x) \triangleright d_1$ and $\ell(o_2) = \mathtt{wr}(x) \triangleright d_2$, then $d_1 \neq d_2$, and there are no operation $\mathtt{wr}(x, 0)$ (which writes the initial value). Let H be a set of labeled posets. We denote by $H_{\neq}$ the subset of differentiated histories of H.

A *renaming* $f : \mathbb{N} \times \mathbb{N}$ is a function which modifies the data values of operations. Given a read/write memory history h, we define by $h[f]$ the history where any number $n \in \mathbb{N}$ appearing in a label of h is changed to $f(n)$.

A set of histories H is *data independent* if, for every history h,

- there exists a differentiated history $h' \in H$, and a renaming f, such that $h = h'[f]$.
- for any renaming f, $h[f] \in H$.

The following lemma shows that for the verification of a data independent set of histories, it is enough to consider differentiated histories.

Lemma 5. *Let H be a data independent set of histories. Then, H is causally consistent (resp.,* CC, CM, CCv*) with respect to the* read/write memory *if and only if $H_{\neq}$ is causally consistent (resp.,* CC, CM, CCv*) with respect to the* read/write memory.

7.2 Characterizing Causal Consistency (CC)

Let $h = (O, \mathsf{PO}, \ell)$ be a differentiated history. We now define and explain the bad patterns of CC. They are defined using the *read-from* relation. The read-from relation relates each write w to each read that reads from w. Since we are considering differentiated histories, we can determine, only by looking at the operations of a history, from which write each read is reading from. There is no ambiguity, as each value can only be written once on each variable.

Definition 4. *The* read-from *relation* RF *is defined as:*

$$\{(o_1, o_2) \mid \exists x \in \mathbb{X}, d \in \mathbb{D}.\ \ell(o_1) = \mathtt{wr}(x, d) \wedge \ell(o_2) = \mathtt{rd}(x) \triangleright d\}.$$

The relation CO *is defined as* $\mathsf{CO} = (\mathsf{PO} \cup \mathsf{RF})^+$.

CyclicCO	there is a cycle in $\mathsf{PO} \cup \mathsf{RF}$ (in CO)
WriteCOInitRead	there is a $\mathtt{rd}(x) \rhd 0$ operation r, and an operation w such that $w <_{\mathsf{CO}} r$ and $\mathsf{var}(w) = \mathsf{var}(r)$
ThinAirRead	there is a $\mathtt{rd}(x) \rhd v$ operation r such that $v \neq 0$, and there is no w operation with $w <_{\mathsf{RF}} r$
WriteCORead	there exist write operations w_1, w_2 and a read operation r_1 in O such that $w_1 <_{\mathsf{CO}} w_2 <_{\mathsf{CO}} r_1$, $w_1 <_{\mathsf{RF}} r_1$, and $\mathsf{var}(w_1) = \mathsf{var}(w_2)$
WriteHBInitRead	there is a $\mathtt{rd}(x) \rhd 0$ operation r, and an operation w such that $w <_{\mathsf{HB}_o} r$ and $\mathsf{var}(w) = \mathsf{var}(r)$, for some o, with $r \leq_{\mathsf{PO}} o$
CyclicHB	there is a cycle in HB_o for some $o \in O$
CyclicCF	there is a cycle in $\mathsf{CF} \cup \mathsf{CO}$

Table 2: All bad patterns defined in the paper.

CC	CM	CCv
CyclicCO	CyclicCO	CyclicCO
WriteCOInitRead	WriteCOInitRead	WriteCOInitRead
ThinAirRead	ThinAirRead	ThinAirRead
WriteCORead	WriteCORead	WriteCORead
	WriteHBInitRead	CyclicCF
	CyclicHB	

Table 3: Bad patterns for each criteria.

Remark 1. *Note that we use lower-case co for the existentially quantified causality order which appears in the definition of causal consistency, while we use upper-case* CO *for the relation fixed as* $(\mathsf{PO} \cup \mathsf{RF})^+$. *The relation* CO *represents the smallest causality order possible. We in fact show in the lemmas 6, 7, and 8, that when a history is* CC *(resp.,* CM, CCv*), the causality order co can always be set to* CO.

There are four bad patterns for CC, defined in terms of the RF and CO relations: CyclicCO, WriteCOInitRead, ThinAirRead, WriteCORead (see Table 2).

Example 8. *History (2e) contains bad pattern* WriteCORead. *Indeed,* $\mathtt{wr}(x,1)$ *is causally related (through relation* CO*) to* $\mathtt{wr}(x,2)$*, which is causally related to* $\mathtt{rd}(x) \rhd 1$*. Intuitively, this means that the site executing* $\mathtt{rd}(x) \rhd 1$ *is aware of both writes* $\mathtt{wr}(x,1)$ *and* $\mathtt{wr}(x,2)$*, but chose to order* $\mathtt{wr}(x,2)$ *before* $\mathtt{wr}(x,1)$*, while* $\mathtt{wr}(x,1)$ *is causally related to* $\mathtt{wr}(x,2)$*. As a result, History (2e) is not* CC *(nor* CM*, nor* CCv*).*

History (2d) contains none of the bad patterns defined in Table 2, and satisfies all definitions of causal consistency. In particular, History (2d) is CC.

Lemma 6. *A differentiated history h is* CC *with respect to* S_{RW} *if and only if h does not contain one of the following bad patterns:* CyclicCO, WriteCOInitRead, ThinAirRead, WriteCORead.

Proof. Let $h = (O, \mathsf{PO}, \ell)$ be a differentiated history.

($\Rightarrow$) Assume that h is CC with respect to S_{RW}. We prove by contradiction that h cannot contain bad patterns CyclicCO, WriteCOInitRead, ThinAirRead, WriteCORead.

First, we show that $\mathsf{CO} \subseteq co$. Given the specification of rd's, and given that h is differentiated, we must have $\mathsf{RF} \subseteq co$. Moreover, by axiom AxCausal, $\mathsf{PO} \subseteq co$. Since co is a transitive order, we thus have $(\mathsf{PO} \cup \mathsf{RF})^+ \subseteq co$ and $\mathsf{CO} \subseteq co$.

(CyclicCO) Since co is acyclic, and $\mathsf{CO} \subseteq co$, CO is acyclic as well.

(WriteCOInitRead) If there is a $\mathtt{rd}(x) \rhd 0$ operation r, and an operation w such that $w <_{\mathsf{CO}} r$ with $\mathsf{var}(w) = x$: we obtain a contradiction, because CC ensures that there exists a sequence $\rho_r \in S_{\mathsf{RW}}$ that orders w before r. However, this is not allowed by S_{RW}, as h is differentiated, and does not contain operation that write the initial value 0. A read $\mathtt{rd}(x) \rhd 0$ can thus happen only when there are no previous write operation on x.

(ThinAirRead) Similarly, we cannot have a $\mathtt{rd}(x) \rhd v$ operation r such that $v \neq 0$, and such that there is no w operation with $w <_{\mathsf{RF}} r$. Indeed, CC ensures that there exists a sequence $\rho_r \in S_{\mathsf{RW}}$ that contains r. Moreover, S_{RW} allows $\mathtt{rd}(x) \rhd v$ operations only when there is a previous write $\mathtt{wr}(x,v)$. So there must exist a $\mathtt{wr}(x,v)$ operation w (such that $w <_{\mathsf{RF}} r$).

(WriteCORead) If there exist $w_1, w_2, r_1 \in O$ such that
- $w_1 <_{\mathsf{RF}} r_1$ and
- $w_1 <_{\mathsf{CO}} w_2$ with $\mathsf{var}(w_1) = \mathsf{var}(w_2)$ and
- $w_2 <_{\mathsf{CO}} r_1$.

Let $x \in \mathbb{X}$ and $d_1 \neq d_2 \in \mathbb{N}$ such that:
- $\ell(w_1) = \mathtt{wr}(x, d_1)$,
- $\ell(w_2) = \mathtt{wr}(x, d_2)$,
- $\ell(r_1) = \mathtt{rd}(x) \rhd d_1$.

By CC, and since $\mathsf{CO} \subseteq co$, we know there exists $\rho_{r_1} \in S_{\mathsf{RW}}$ that contains w_1 before w_2, and ends with r_1. The specification S_{RW} require the last write operation on x to be a $\mathtt{wr}(x, d_1)$. However, as h is differentiated, the only $\mathtt{wr}(x, d_1)$ operation is w_1. As a result, the last write operation on variable x in ρ_{r_1} cannot be w_1 (as w_2 is after w_1), and we have a contradiction.

($\Leftarrow$) Assume that h contains none of the bad patterns described above. We show that h is CC. We use for this the strict partial order $\mathsf{CO} = (\mathsf{PO} \cup \mathsf{RF})^+$ as the causal order co. The relation CO is a strict partial order, as h does not contain bad pattern CyclicCO. Axiom AxCausal holds by construction. We define for each operation $o \in O$ a sequence $\rho_o \in S_{\mathsf{RW}}$ such that $\mathsf{CausalHist}(o)\{o\} \leq \rho_o$ (such that AxCausalValue holds).

Let $o \in O$. We have three cases to consider.

1) If o is a wr operation, then all the return values of the read operations in $\mathsf{CausalHist}(o)\{o\}$ are hidden. We can thus define ρ_o as any sequentialization of $\mathsf{CausalHist}(o)\{o\}$, and where we add the appropriate return values to the read operations (the value written by the last preceding write on the same variable).

2) If o is a rd operation r, labeled by $\mathtt{rd}(x) \rhd 0$ for some $x \in \mathbb{X}$. We know by the fact that h does not contain WriteCOInitRead that there is no w such that $w <_{\mathsf{CO}} r$. As a result, we can define ρ_o as any sequentialization of $\mathsf{CausalHist}(o)\{o\}$, where we add the appropriate return values to the read operations different from r.

3) If o is a rd operation r_1, labeled by $\mathtt{rd}(x) \rhd d_1$ for some $x \in \mathbb{X}$ and $d_1 \neq 0$, we know by assumption that there exists a wr operation w_1 such that $w_1 <_{\mathsf{RF}} r_1$ (h does not contain ThinAirRead).

We also know (h does not contain WriteCORead) there is no w_2 such that
- $w_1 <_{\mathsf{RF}} r_1$ and
- $w_1 <_{\mathsf{CO}} w_2$ with $\mathsf{var}(w_1) = \mathsf{var}(w_2)$ and
- $w_2 <_{\mathsf{CO}} r_1$.

This ensures that w_1 must be a maximal wr operation on variable x in $\mathsf{CausalHist}(o)$. It is thus possible to sequentialize $\mathsf{CausalHist}(o)\{o\}$ into ρ_o, so that w_1 is the last write on variable x. We can then add appropriate return values to the read operations different than r_1, whose return values were hidden in $\mathsf{CausalHist}(o)\{o\}$ (the value written by the last preceding write in ρ_o on the same variable). ☐

7.3 Characterizing Causal Convergence (CCv)

CCv is stronger than CC. Therefore, CCv excludes all the bad patterns of CC, given in Lemma 6. CCv also excludes one additional bad pattern, defined in terms of a *conflict relation*.

The *conflict relation* is a relation on write operations (which write to the same variable). It is used for the bad pattern CyclicCF of CCv, defined in Table 2. Intuitively, for two write operations w_1 and w_2, we have $w_1 <_{\mathsf{CF}} w_2$ if some site saw both writes, and decided to order w_1 before w_2 (so decided to return the value written by w_2).

Example 9. *History (2a) contains bad pattern* CyclicCF*. The* $\mathtt{wr}(x,1)$ *operation* w_1 *is causally related to the* $\mathtt{rd}(x) \triangleright 2$ *operation, so we have* $w_1 <_{\mathsf{CF}} w_2$*, where* w_2 *is the* $\mathtt{wr}(x,2)$ *operation. Symmetrically,* $w_2 <_{\mathsf{CF}} w_1$*, and we obtain a cycle. On the other hand, History (2a) does not contain any of the bad patterns of* CM*.*

Example 10. *History (2c) contains bad pattern* CyclicCF*. The cycle is on the two writes operations* $\mathtt{wr}(x,1)$ *and* $\mathtt{wr}(x,2)$*.*

The formal definition of the conflict relation is the following.

Definition 5. *We define the* conflict relation $\mathsf{CF} \subseteq O \times O$ *to be the smallest relation such that: for all* $x \in \mathbb{X}$*, and* $d_1 \neq d_2 \in \mathbb{N}$ *and operations* w_1, w_2, r_2*, if*

- $w_1 <_{\mathsf{CO}} r_2$,
- $\ell(w_1) = \mathtt{wr}(x,d_1)$,
- $\ell(w_2) = \mathtt{wr}(x,d_2)$, *and*
- $\ell(r_2) = \mathtt{rd}(x) \triangleright d_2$,

then $w_1 <_{\mathsf{CF}} w_2$*.*

We obtain the following lemma for the bad patterns of CCv.

Lemma 7. *A differentiated history* h *is* CCv *with respect to* S_{RW} *if and only if* h *is* CC *and does not contain the following bad pattern:* CyclicCF*.*

7.4 Characterizing Causal Memory (CM)

CM is stronger than CC. Therefore, CM excludes all the bad patterns of CC, given in Lemma 6. CM also excludes two additional bad patterns, defined in terms of a *happened-before relation*.

The *happened-before relation for an operation* $o \in O$ intuitively represents the minimal constraints that must hold in a sequence containing all operations before o, on the site of o.

Example 11. *History (2b) contains bad pattern* WriteHBInitRead*. Indeed, we have* $\mathtt{wr}(z,1) <_{\mathsf{PO}} \mathtt{wr}(x,1) <_{\mathsf{HB}_{r_2}} \mathtt{wr}(x,2) <_{\mathsf{PO}} \mathtt{rd}(z) \triangleright 0$*, where* r_2 *is the* $\mathtt{rd}(x) \triangleright 2$ *operation. The edge* $\mathtt{wr}(x,1) <_{\mathsf{HB}_{r_2}} \mathtt{wr}(x,2)$ *is induced by the fact that* $\mathtt{wr}(x,1) <_{co} \mathtt{rd}(x) \triangleright 2$*.*

The formal definition of HB_o is the following.

Definition 6. *Given* $o \in O$*, we define the* happened-before relation for o*, noted* $\mathsf{HB}_o \subseteq O \times O$*, to be the smallest relation such that:*

- $\mathsf{CO}_{|\mathsf{CausalPast}(o)} \subseteq \mathsf{HB}_o$, *and*
- HB_o *is transitive, and*
- *for* $x \in \mathbb{X}$*, and* $d_1 \neq d_2 \in \mathbb{N}$*, if*
  - $w_1 <_{\mathsf{HB}_o} r_2$,
  - $r_2 \leq_{\mathsf{PO}} o$,
  - $\ell(w_1) = \mathtt{wr}(x,d_1)$,
  - $\ell(w_2) = \mathtt{wr}(x,d_2)$, *and*
  - $\ell(r_2) = \mathtt{rd}(x) \triangleright d_2$,

  then $w_1 <_{\mathsf{HB}_o} w_2$*.*

There are two main differences with the conflict relation CF. First, CF is not defined inductively in terms of itself, but only in terms of the relation CO. Second, in the happened-before relation for o, in order to add an edge between write operations, there is the constraint that $r_2 \leq_{\mathsf{PO}} o$, while in the definition of the conflict relation, r_2 is an arbitrary read operation. These differences make the conflict and happened-before relations not comparable (with respect to set inclusion).

We obtain the following lemma for the bad patterns of CM (see Table 2 for the bad patterns' definitions).

Lemma 8. *A differentiated history* h *is* CM *with respect to* S_{RW} *if and only if* h *is* CC *and does* not *contain the following bad patterns:* WriteHBInitRead, CyclicHB.

Table 3 gives, for each consistency criterion, the bad patterns which are excluded by the criterion.

8. Single History Consistency under DI

The lemmas of the previous sections entail a polynomial-time algorithm for checking whether a given differentiated history is causally consistent (for any definition). This contrasts with the fact that checking whether an arbitrary history is causally consistent is NP-complete.

The algorithm first constructs the relations which are used in the definitions of the bad patterns, and then checks for the presence of the bad patterns in the given history.

Lemma 9. *Let* $h = (O, \mathsf{PO}, \ell)$ *be a differentiated history. Computing the relations* RF,CO,CF*, and* HB_o *for* $o \in O$ *can be done in polynomial time (*$O(n^5)$ *where* n *is the number of operations in* h*).*

Proof. We show this for the relation HB_o (for some $o \in O$). The same holds for the other relations. The relation HB_o can be computed inductively using its fixpoint definition. At each iteration of the fixpoint computation, we add one edge between operations in O. Thus, there are at most n^2 iterations.

Each iteration takes $O(n^3)$ time. For instance, an iteration of computation of HB_o can consist in adding an edge by transitivity, which takes $O(n^3)$ time.

Thus the whole computation of HB_o takes $O(n^5)$ time. □

Once the relations are computed, we can check for the presence of bad patterns in polynomial time.

Theorem 3. *Let* $h = (O, \mathsf{PO}, \ell)$ *be a differentiated history. Checking whether* h *is* CC *(resp.,* CM*, resp.,* CCv*) can be done in polynomial time (*$O(n^5)$ *where* n *is the number of operations in* h*).*

Proof. First, we compute the relations $\mathsf{RF}, \mathsf{CO}, \mathsf{CF}, \mathsf{HB}_o$ (for all $o \in O$), in time $O(n^5)$. The presence of bad patterns can be checked in polynomial time. For instance, for bad pattern CyclicCF, we need to find a cycle in the relation CF. Detecting the presence of a cycle in a relation takes $O(n^2)$.

The complexity of the algorithm thus comes from computing the relations, which is $O(n^5)$. □

In the next two sections, we only consider criterion CC.

9. Reduction to Control-State Reachability under Data Independence

The undecidability proof of Theorem 2 uses an implementation which is not data independent. Therefore, it does not apply when we consider only data independent implementations. In fact, we show that for read/write memory implementations which are data independent, there is an effective reduction from checking CC to a non-reachability problem.

Using the characterization of Section 7.2, we define an *observer* $\mathcal{M}_{\mathsf{CC}}$ that looks for the bad patterns leading to non-CC. More precisely, our goal is to define $\mathcal{M}_{\mathsf{CC}}$ as a *register automaton* such that (by an abuse of notation, the set of executions recognized by $\mathcal{M}_{\mathsf{CC}}$ is also denoted $\mathcal{M}_{\mathsf{CC}}$):

$$\mathcal{I} \text{ is } \mathtt{CC} \text{ with respect to } S_{\mathsf{RW}} \iff \mathcal{I} \cap \mathcal{M}_{\mathsf{CC}} = \varnothing$$

where $\mathcal{I}$ is any data independent implementation.

Ultimately, we exploit in Section 10 this reduction to prove that checking CC for (finite-state) data independent implementations, with respect to the read/write memory specification, is decidable.

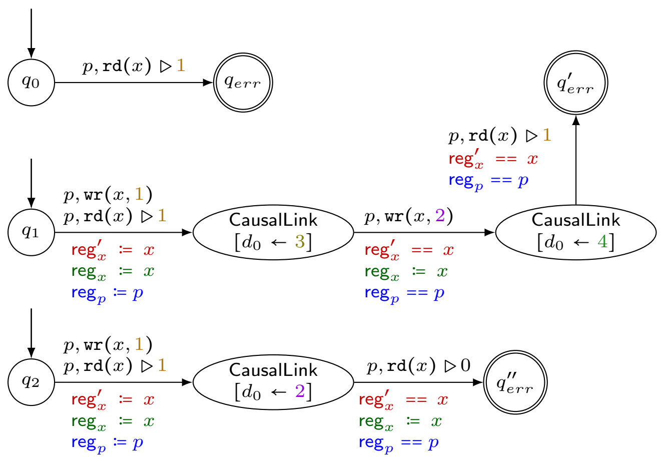

Figure 4: The observer $\mathcal{M}_{\mathsf{CC}}$, finding bad patterns for CC with respect to S_{RW}. The first branch looks for bad pattern ThinAirRead. The second branch looks for bad pattern WriteCORead. The third branch looks for bad pattern WriteCOInitRead.
Each state has a self-loop with any symbol containing value 5, which we do not represent. Two labels $p, m(arg) \triangleright rv$ above a transition denote two different transitions.

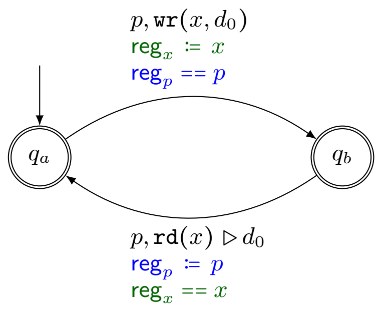

Figure 5: The register automaton CausalLink, which recognizes causality chains by following links in the PO ∪ RF relations. Both states are final.
Each state has a self-loop with any symbol containing value 5, which we do not represent.

9.1 Register Automata

Register automata [11] have a finite number of registers in which they can store values (such as the site identifier, the name of a variable in the read/write memory, or the data value stored at a particular variable), and test equality on stored registers.

We describe the syntax of register automata that we use in the figures. The label $p, \mathtt{wr}(x, 1)$ above the transition going from q_1 in Figure 4 is a form of pattern matching. If the automaton reads a tuple $(p_0, \mathtt{wr}(x_0, 1))$, for some $p_0 \in \mathsf{PId}$, $x_0 \in \mathbb{X}$, then the variables p, x are bound respectively to p_0, x_0.

If this transition, or another transition, gets executed afterwards, the variables p, x can be bound to other values. These variables are only local to a specific execution of the transition.

The instruction $\mathsf{reg}'_x := x$, on this same transition, is used to store the value x_0 which was bound by x in register reg'_x. This ensures that the operations $\mathtt{wr}(x, 2)$ and $\mathtt{rd}(x) \triangleright 1$ that come later use the same variable x_0, thanks to the equality check $\mathsf{reg}'_x == x$.

Note that, in Figure 5, d_0 is not a binding variable as p and x, but is instead a constant which is fixed to different values in Figure 4.

9.2 Reduction

$\mathcal{M}_{\mathsf{CC}}$ (see Figure 4) is composed of three parts. The first part recognizes executions which contain bad pattern ThinAirRead, i.e. which have a rd operation with no corresponding wr. The second part recognizes executions containing bad pattern WriteCORead, composed of operations w_1, w_2, and r_1, such that $w_1 <_{\mathsf{CO}} w_2 <_{\mathsf{CO}} r_1$, $w_1 <_{\mathsf{RF}} r_1$, and $\mathsf{var}(w_1) = \mathsf{var}(w_2)$. The third part recognizes executions containing bad pattern WriteCOInitRead, where a write on some variable x, writing a non-initial value, is causally related to a $\mathtt{rd}(x) \triangleright 0$.

To track the relation CO, $\mathcal{M}_{\mathsf{CC}}$ uses another register automaton, called CausalLink (see Figure 5), which recognizes unbounded chains in the CO relation.

The registers of $\mathcal{M}_{\mathsf{CC}}$ store site ids and the variables' names of the read/write memory (we use registers because the number of sites and variables in the causality links can be arbitrary).

By data independence, $\mathcal{M}_{\mathsf{CC}}$ only needs to use a bounded number of values. For instance, for the second branch recognizing bad pattern WriteCORead, it uses value 1 for operations w_1 and r_1, and value 2 for operation w_2. It uses value 3 for the causal link between w_1 and w_2, and value 4 for the causal link between w_2 and r_1. Finally, it uses the value $5 \in \mathbb{N}$ for all actions of the execution which are not part of the bad pattern. As a result, it can self-loop with any symbol containing value 5. We do not represent these self-loops to keep the figure simple.

We prove in Theorem 4 that any execution recognized by $\mathcal{M}_{\mathsf{CC}}$ is not CC. Reciprocally, we prove that for any differentiated execution of an implementation $\mathcal{I}$ which is not CC, we can rename the values to obtain an execution with 5 values recognized by $\mathcal{M}_{\mathsf{CC}}$. By data independence of $\mathcal{I}$, the renamed execution is still an execution of $\mathcal{I}$.

Remark 2. *Note here that the observer $\mathcal{M}_{\mathsf{CC}}$ does not look for bad pattern* CyclicCO. *We show in Theorem 4 that, since the implementation is a prefix-closed set of executions, it suffices to look for bad pattern* ThinAirRead *to recognize bad pattern* CyclicCO.

Theorem 4. *Let $\mathcal{I}$ be a data independent implementation. $\mathcal{I}$ is* CC *with respect to S_{RW} if and only if $\mathcal{I} \cap \mathcal{M}_{\mathsf{CC}} = \varnothing$.*

This result allows to reuse any tool or technique that can solve reachability (in the system composed of the implementation and the observer $\mathcal{M}_{\mathsf{CC}}$) for the verification of CC (with respect to the read/write memory).

10. Decidability under Data Independence

In this section, we exploit this reduction to obtain decidability for the verification of CC with respect to the read/write memory.

We consider a class of implementations $\mathcal{C}$ for which reachability is decidable, making CC decidable (EXPSPACE-complete). We consider implementations $\mathcal{I}$ which are distributed over a finite number of sites. The sites run asynchronously, and communicate by sending messages using peer-to-peer communication channels.

Moreover, we assume that the number of variables from the read/write memory that the implementation $\mathcal{I}$ seeks to implement is fixed and finite as well. However, we do not bound the domain of values that the variables can store.

Each site is a finite-state machine with registers that can store values in $\mathbb{N}$ for the contents of the variables in the read/write memory. Registers can be assigned using instructions of the form

$x ::= y$ and $x ::= d$ where x, y are registers, and $d \in \mathbb{N}$ is a value (a constant, or a value provided as an argument of a method). Values can also be sent through the network to other sites, and returned by a method. We make no assumption on the network: the peer-to-peer channels are unbounded and unordered.

Any implementation in $\mathcal{C}$ is thus necessarily data independent by construction, as the contents of the registers storing the values which are written are never used in conditionals.

The observer $\mathcal{M}_{\texttt{CC}}$ we constructed only needs 5 values to detect all CC violations. For this reason, when modeling an implementation in $\mathcal{C}$, there is no need to model the whole range of natural numbers $\mathbb{N}$, but only 5 values. With this in mind, any implementation in $\mathcal{C}$ can be modelled by a Vector Addition System with States (VASS) [25, 27], or a Petri Net [16, 37]. The local state of each site is encoded in the state of the VASS, and the content of the peer-to-peer channels is encoded in the counters of the VASS. Each counter counts how many messages there are of a particular kind in a particular peer-to-peer channel. There exist similar encodings in the literature [8].

Then, since the number of sites and the number of variables is bounded, we can get rid of the registers in the register automaton of the observer $\mathcal{M}_{\texttt{CC}}$, and obtain a (normal) finite automaton. We then need to solve control-state reachability in the system composed of the VASS and the observer $\mathcal{M}_{\texttt{CC}}$ to solve CC (according to Theorem 4). Since VASS are closed under composition with finite automata, and control-state reachability is EXPSPACE-complete for VASS, we get the EXPSPACE upper bound for the verification of CC (for the read/write memory).

The EXPSPACE lower bound follows from: (1) State reachability in class $\mathcal{C}$ is EXPSPACE-complete, equivalent to control-state reachability in VASS [4]. Intuitively, a VASS can be modelled by an implementation in $\mathcal{C}$, by using the unbounded unordered channels to simulate the counters of the VASS. (Similar to the reduction from $\mathcal{C}$ to VASS outlined above, but in the opposite direction.) (2) Checking reachability can be reduced to verifying CC. Given an implementation $\mathcal{I}$ in $\mathcal{C}$, and a state q, knowing whether q is reachable can be reduced to checking whether a new implementation $\mathcal{I}'$ is not causally consistent. $\mathcal{I}'$ is an implementation which simulates $\mathcal{I}$, and produces only causally consistent executions; if it reaches state q, it artificially produces a non-causally consistent execution, for instance by returning wrong values to read requests.

Theorem 5. *Let $\mathcal{I}$ be a data independent implementation in $\mathcal{C}$. Verifying* CC *of $\mathcal{I}$ with respect to the* read/write memory *is* EXPSPACE*-complete (in the size of the VASS of $\mathcal{I}$).*

11. Related Work

Wei et al. [42] studied the complexity of verifying PRAM consistency (also called FIFO consistency) for one history. They proved that the problem is NP-complete. For differentiated executions, they provided a polynomial-time algorithm.

Independently, Furbach et al. [19] showed that checking causal consistency (CM definition) of one history is an NP-complete problem. They proved that checking consistency for one history for any criterion stronger than SLOW consistency and weaker than sequential consistency is NP-complete, where SLOW consistency ensures that for each variable x, and for each site p, the reads of p on variable x can be explained by ordering all the writes to x while respecting the program order. This range covers CM, but does not cover CC (see Figure 2c for a history which is CC but not SLOW). It is not clear whether this range covers CCv. To prove NP-hardness, they used a reduction from the NP-complete SAT problem. We show that their encoding can be reused to show NP-hardness for checking whether a history is CC (resp., CCv) with respect to the read/write memory specification.

Concerning verification, we are not aware of any work studying the decidability or complexity of checking whether all executions of an implementation are causally consistent. There have been works on studying the problem for other criteria such as linearizability [24] or eventual consistency [40]. In particular, it was shown that checking linearizability is an EXPSPACE-complete problem when the number of sites is bounded [3, 22]. Eventual consistency has been shown to be decidable [8]. Sequential consistency was shown to be undecidable [3].

The approach we adopted to obtain decidability of causal consistency by defining bad patterns for particular specifications has been used recently in the context of linearizability [1, 9, 23]. However, the bad patterns for linearizability do not transfer to causal consistency. Even from a technical point of view, the results introduced for linearizability cannot be used in our case. One reason for this is that, in causal consistency, there is the additional difficulty that the causal order is existentially quantified, while the happens-before relation in linearizability is fixed (by a global clock).

Lesani et al. [32] investigate mechanized proofs of causal consistency using the theorem prover Coq. This approach does not lead to full automation however, e.g., by reduction to assertion checking.

12. Conclusion

We have shown that verifying causal consistency is hard, even undecidable, in general: verifying whether one single execution satisfies causal consistency is NP-hard, and verifying if all the executions of an implementation are causally consistent is undecidable. These results are not due to the complexity of the implementations nor of the specifications: they hold even for finite-state implementations and specifications. They hold already when the specification corresponds to the simple read-write memory abstraction. The undecidability result contrasts with known decidability results for other correctness criteria such as linearizability [3] and eventual consistency [8].

Fortunately, for the read-write memory abstraction, an important and widely used abstraction in the setting of distributed systems, we show that, when implementations are data-independent, which is the case in practice, the verification problems we consider become tractable. This is based on the very fact that data independence allows to restrict our attention to differentiated executions, where the written values are unique, which allows to deterministically establish the read-from relation along executions. This is crucial for characterizing by means of a finite number of bad patterns the set of all violations to causal consistency, which is the key to our complexity and decidability results for the read-write memory.

First, using this characterization we show that the problem of verifying the correctness of a single execution is polynomial-time in this case. This is important for building efficient and scalable testing and bug detection algorithms. Moreover, we provide an algorithmic approach for verifying causal consistency (w.r.t. the read-write memory abstraction) based on an effective reduction of this problem to a state reachability problem (or invariant checking problem) in the class of programs used for the implementation. Regardless from the decidability issue, this reduction holds for an unbounded number of sites (in the implementation), and an unbounded number of variables (in the read-write memory). In fact, it establishes a fundamental link between these two problems and allows to use all existing (and future) program verification methods and tools for the verification of causal consistency. In addition, when the number of sites is bounded, this reduction provides a decidability result for verifying causal consistency concerning a significant class of implementations: finite-control machines (one per site) with data registers (over an unrestricted data domain, with only assignment operations and equality testing), communicating through unbounded unordered channels. As far as we know, this is the first work that establishes complexity and (un)decidability results for the verification of causal consistency.

All our results hold for the three existing variants of causal consistency CC, CM, and CCv, except for the reduction to state reachability and the derived decidability result that we give in this paper for CC only. For the other two criteria, building observers detecting their corresponding bad patterns is not trivial in general, when there is no assumption on the number of sites and the number of variables (in the read-write memory). We still do not know if this can be done using the same class of state-machines we use in this paper for the observers. However, this can be done if these two parameters are bounded. In this case, we obtain a decidability result that holds for the same class of implementations as for CC, but this time for a fixed number of variables in the read-write memory. This is still interesting since when data independence is not assumed, verifying causal consistency is undecidable for the read-write memory even when the number of sites is fixed, the number of variables is fixed, and the data domain is finite. We omit these results in this paper.

Finally, let us mention that in this paper we have considered correctness criteria that correspond basically to safety requirements. Except for CCv, convergence, meaning eventual agreement between the sites on their execution orders of non-causally dependent operations is not guaranteed. In fact, these criteria can be strengthened with a liveness part requiring the convergence property. Then, it is possible to extend our approach to handle the new criteria following the approach adopted in [8] for eventual consistency. Verifying correctness in this case can be reduced to a repeated reachability problem, and model-checking algorithms can be used to solve it.

For future work, it would be very interesting to identify a class of specifications for which our approach is systematically applicable, i.e., for which there is a procedure producing the complete set of bad patterns and their corresponding observers in a decidable class of state machines.

Acknowledgments

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), and by an EPFL-Inria postdoctoral grant.

References

[1] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS '13*. Springer, 2013.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2), 2000.

[4] M. F. Atig, A. Bouajjani, and T. Touili. Analyzing asynchronous programs with preemption. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*, pages 37–48, 2008.

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD '13*, pages 761–772, New York, NY, USA, 2013. ACM.

[6] L. Benmouffok, J.-M. Busca, J. M. Marquès, M. Shapiro, P. Sutra, and G. Tsoukalas. Telex: A semantic platform for cooperative application development. In *CFSE*, Toulouse, France, 2009.

[7] K. P. Birman. *Replication and fault-tolerance in the ISIS system*, volume 19. ACM, 1985.

[8] A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *POPL '14*, pages 285–296, New York, NY, USA, 2014. ACM.

[9] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. In *ICALP '15*, pages 95–107. Springer, 2015.

[10] A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza. On Verifying Causal Consistency. *ArXiv e-prints*, Nov. 2016.

[11] P. Bouyer, A. Petit, and D. Thérien. An algebraic characterization of data and timed languages. In K. G. Larsen and M. Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 2001.

[12] S. Burckhardt. *Principles of Eventual Consistency*. now publishers, October 2014.

[13] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft Research.

[14] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: scalable causal consistency using dependency matrices and physical clocks. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 11:1–11:14, 2013.

[15] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014*, pages 4:1–4:13, 2014.

[16] J. Esparza. Decidability and complexity of petri net problems — an introduction. In *Lectures on Petri Nets I: Basic Models*. Springer Berlin Heidelberg, 1998.

[17] C. J. Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science, 1987.

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[19] F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. Memory model-aware testing - a unified complexity analysis. In *Application of Concurrency to System Design (ACSD), 2014 14th International Conference on*, pages 92–101, June 2014. doi: 10.1109/ACSD.2014.27.

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

[21] J. Hamza. *Algorithmic Verification of Concurrent and Distributed Data Structures*. PhD thesis, Université Paris Diderot, 2015.

[22] J. Hamza. On the complexity of linearizability. In *NETYS '15*, volume 9466 of *Lecture Notes in Computer Science*. Springer, 2015.

[23] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR '13*. Springer, 2013.

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

[25] J. Hopcroft and J.-J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8(2):135–159, 1979.

[26] E. Jiménez, A. Fernández, and V. Cholvi. A parametrized algorithm that implements sequential, causal, and cache memory consistencies. *J. Syst. Softw.*, 81(1):120–131, Jan. 2008. ISSN 0164-1212.

[27] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and system Sciences*, 3(2):147–195, 1969.

[28] A.-M. Kermarrec, A. I. T. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *PODC*, pages 210–218, 2001.

[29] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4): 360–391, Nov. 1992. ISSN 0734-2071.

[30] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[31] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979. ISSN 0018-9340.

[32] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: certified causally consistent distributed key-value stores. In *ACM SIGPLAN Notices*, volume 51, pages 357–370. ACM, 2016.

[33] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, pages 401–416, 2011.

[34] F. Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.

[35] J. Michaux, X. Blanc, M. Shapiro, and P. Sutra. A semantically rich approach for collaborative model edition. In *SAC*, pages 1470–1475, 2011.

[36] M. Perrin, A. Mostefaoui, and C. Jard. Causal consistency: Beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 26:1–26:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4092-2.

[37] C. A. Petri. Kommunikation mit automaten. 1962.

[38] M. Raynal and A. Schiper. From causal consistency to sequential consistency in shared memory systems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 180–194. Springer, 1995.

[39] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-6400-2.

[40] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In M. B. Jones, editor, *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 172–183. ACM, 1995.

[41] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8.

[42] H. Wei, Y. Huang, J. Cao, X. Ma, and J. Lu. Verifying PRAM consistency over read/write traces of data replicas. *CoRR*, abs/1302.5161, 2013.

[43] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL '86: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–193. ACM Press, 1986.

[44] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3618-5.