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 [29] (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, 13, 14, 25, 32], contrary to strong consistency (linearizability) which cannot be ensured in the presence of network partitions and while the system remains available [17, 19] (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 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 [16, 33]. 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.
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 linearity (stronger than CC) [23] and eventual consistency (weaker than CC) [39] are decidable to verify in that same setting [3, 8, 21]. 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 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 \( o_2 \) 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, 42]. 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 [10], i.e., finite-state state machines equipped 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., [11, 12, 20, 35, 37]). In this paper we consider three important variants. The variant called simply causal consistency (abbreviated as CC) allows non-causally depen-
dent 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, 27, 34, 39]. We also consider two stronger notions, namely causal memory (CM) [2, 35], and causal convergence (CCv) [11, 12, 35]. 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 in a variety of systems [5, 14, 28, 38, 40, 43] 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 [35].

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 regular specifications (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. the RWM to a state reachability (or dually to an invariant checking) problem (Section 9).

2. Notations

2.1 Sets, Multisets, Relations

Given a set O and a relation R ⊆ O × O, we denote by o1 <o R o2 the fact that (o1, o2) ∈ R. We denote by o1 ≤o R o2 the fact that o1 <o R o2 or o1 = o2. We denote by R+ the transitive closure of R, which is the composition of one or more copies of R.

Let O′ be a subset of O. Then R|O′ is the relation R projected on the set O′, that is, {(o1, o2) ∈ R | o1, o2 ∈ O′}. The set O′ ⊆ O is said to be downward-closed (with respect to relation R) if ∀o1, o2, if o2 ∈ O′ and o1 ≤o R o2, then o1 ∈ O′ as well.

Figure 1: Implication graph of causal consistency definitions.

2.2 Labeled Posets

A relation < ⊆ O × 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, <, ℓ) where (O, <) is a poset and ℓ : O → Σ is the labeling function.

We say that ρ′ is a prefix of ρ if there exists a downward closed set A ⊆ O (with respect to relation <) such that ρ′ = (A, <, ℓ′). (resp., labeled) sequential poset (sequence for short) is a (resp., labeled) poset where the relation < is a strict total order. We denote by ℓ ◦ ℓ′ 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.

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 M with input or output values from a domain D.

For instance, in the case of the read/write memory, the set of methods M is {var, rd} for writing or reading a variable. Also, given a set of variables X, the domain D is defined as (X × N) ∪ X ∪ N ∪ {⊥}. Write operations take as input a variable in X and a value in N and return ⊥ while read operations take as input a variable in X and return a value in N. The return value ⊥ is often omitted for better readability.

A history h = (O, PO, ℓ) is a poset labeled by M×D×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 o1, o2 ∈ O, o1 ≤po o2 means that o1 and o2 were submitted to the same site, and o1 occurred before o2,
- for m ∈ M and arg, rv ∈ D, and o ∈ O, ℓ(o) = (m, arg, rv) means that operation o is an invocation of m with input arg and returning rv.

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

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 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 $(x \in X$ and $v \in N)$:

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

4. Causal Consistency

Causal consistency is one of the most widely used consistency criteria 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 [20, 35] ($CC$ 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 $\text{co}$ 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, $\text{wr}(x,1)$ must be causally related to $\text{rd}(y) \triangleright 1$ (so that the read can return value 1). By transitivity of the causality order and because any causality order must contain the program order, $\text{wr}(x,1)$ must be causally related to $\text{wr}(x,2)$. However, site $p_0$ first reads $\text{rd}(x) \triangleright 2$, and then $\text{rd}(x) \triangleright 1$. 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 $\text{wr}(x,1)$ and $\text{wr}(x,2)$ are not causally related to each other but each write is causally related to both reads. Since the writes are not causally related, site $p_0$ can read them in any order.

There is a subtlety here. In History (2c), site $p_0$ first does $\text{rd}(x) \triangleright 1$, which implicitly means that it executed $\text{wr}(x,1)$ after $\text{wr}(x,2)$. Then $p_0$ does $\text{rd}(x) \triangleright 2$ which means that $p_0$ “changed its mind”, and decided to order $\text{wr}(x,2)$ after $\text{wr}(x,1)$. This is

---

In general, specifications can be defined as sets of posts 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.

---

Axiom $\text{AxCausal}$ states that the causal order must at least contain the program order. Axiom $\text{AxCausalValue}$ states that, for each operation $o \in O$, the causal history of $o$ (roughly, all the operations allowed by $CC$, but as we will see later, not allowed by the stronger criteria $CM$ and $CCv$.

This feature of $CC$ is useful for systems which do speculative executions and rollbacks [6, 39]. 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 $h = (O, \text{PO}, \ell)$ is $CC$ with respect to a specification $S$ when there exists a strict partial order $\text{co} \subseteq O \times O$, called causal order, such that, for all operations $o \in O$, there exists a specification sequence $\rho_o \in S$ such that axioms $\text{AxCausal}$ and $\text{AxCausalValue}$ hold (see Table 1).

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

---

Table 1: Axioms used in the definitions of causal consistency.
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 \( o \), \text{CausalPast}(o)\), as the set of operations before \( o \) in the causal order and the causal history of \( o \), \text{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 \text{AxCausalValue} uses the causal history where only the return values of operation \( o \) have been kept. This is denoted by \( \text{CausalHist}(o)(o) \). The fact that the latter can be sequentialized to a sequence \( \rho_s \) in the specification is denoted by \text{CausalHist}(o)(\rho_s) \leq \rho_o\). We defer the formal definition of these two last notions 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 \( \rho = (O, \prec, \ell) \) be a \( M \times D \times D \) 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 \( (M \times D) \cup (M \times D \times D) \) labeled poset \((O', \prec, \ell')\) where for all \( o \in O' \), \( \ell'(o) = \ell(o) \), and for all \( o \in O \setminus O' \), if \( \ell(o) = (m, \text{arg}, \text{rv}) \), then \( \ell'(o) = (m, \text{arg}) \). If \( O' = \{o\} \), we denote \( \rho(O') \) by \( \rho(o) \).

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

\[
\begin{align*}
\ell'(o) &\leq \ell(o), \\
\ell'(o) &\leq (m, \text{arg}, \text{rv}) \Rightarrow \ell'(o) = (m, \text{arg}).
\end{align*}
\]

**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 \( \rho \), but some return values are hidden in \( \rho(O') \).

**Example 4.** In Figure 3, we have \( \rho_o \leq \rho_o \), as the only differences between \( \rho_o \) and \( \rho_o \) is the label of \( c_1 \), and the fact that \( o_1 < o_2 \) holds in \( \rho_o \) but not in \( \rho_o \).

We have \( \rho_o \not\leq \rho_o \), as \( o_1 < o_1 \) holds in \( \rho_o \), but not in \( \rho_o \).

### 4.4 Causal Memory (CM)

Compared to causal consistency, causal memory [2, 35] (denoted \( \text{CM} \)) does not allow a site to “change its mind” about the order of the operations. The original definition of causal memory of Ahamad et al. [2] applies only to the read/write memory and it was extended by Perrin et al. [35] 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 \( \text{CC} \) but not \( \text{CM} \). Intuitively, the reason is that site \( p_o \) first decides to order \( \text{wr}(x, 1) \) after \( \text{wr}(x, 2) \) (for \( \text{rd}(x) \succ 1 \)) and then decides to order \( \text{wr}(x, 2) \) after \( \text{wr}(x, 1) \) (for \( \text{rd}(x) \succ 2 \)).

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

Formally, a history \( h = (O, PO, \ell) \) is \( \text{CM} \) 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 \( \text{AxCausal} \) and \( \text{AxCausal} \) hold. With respect to \( \text{CC} \), 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, \( \text{AxCausalSeq} \) states:

\[
\text{CausalHist}(o)(\text{POPast}(o)) \leq \rho_o
\]

where \( \text{CausalHist}(o)(\text{POPast}(o)) \) is the causal history where only the return values of the operations which are before \( o \) in the program order (in \( \text{POPast}(o) \)) 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 site \( p_h \) in History (2c).

The following lemma gives the relationship between \( \text{CM} \) and \( \text{CC} \).

**Lemma 1 (\cite{35}).** If a history \( h \) is \( \text{CM} \) with respect to a specification \( S \), then \( h \) is \( \text{CC} \) 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 \( \text{AxCausal} \) and \( \text{AxCausal} \). In particular, for any \( o \in O \), we have \( \text{CausalHist}(o)(\text{POPast}(o)) \leq \rho_o \).

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

### 4.5 Causal Convergence (CCv)

Our formalization of causal convergence (denoted \( \text{CCv} \)) corresponds to the definition of causal consistency given in Burckhardt et al. [12] and Burckhardt [11] restricted to sequential specifications. \( \text{CCv} \) 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 \( \not\text{causally related} \). As it was the case for the causal order, the arbitration order, denoted by \( \text{arb} \), is not encoded explicitly in the notion of history and it is existentially quantified in the definition of \( \text{CCv} \).

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

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

On the other hand, History (2b) is not \( \text{CM} \). If History (2b) were \( \text{CM} \), site \( p_o \), \( \text{wr}(y, 1) \) should go before \( \text{rd}(y) \succ 1 \). By transitivity, this implies that \( \text{wr}(x, 1) \) should go before \( \text{rd}(x) \succ 2 \). But for the
read $rd(x) \triangleright 2$ to return value 2. $wr(x, 1)$ should then also go before $wr(x, 2)$. This implies that $wr(z, 1)$ goes before $rd(z) \triangleright 0$ preventing $rd(z) \triangleright 0$ 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 [30] 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 $wr(x, 1)$ and $wr(x, 2)$, and the final reads $rd(x) \triangleright 1$ and $rd(x) \triangleright 2$, all the operations of $p_o$ must be completely ordered before the operations of $p_o$, or vice versa. This would make one of the $rd(y) \triangleright 0$ to be ordered after either $wr(y, 1)$ or $wr(y, 2)$, 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 $p_o$ to $p_o$ from $p_o$ to $p_o$.

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 $p_o \in S$ 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 ([35]). 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 $CausalHist(o)(o) \leq CausalArb(o)(o)$ (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 $S_{eq}$ 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 $p_o$ for each operation $o$, and then check in polynomial time whether the axioms of Table 1 hold, and whether $p_o \in S$.

For NP-hardness, we reduce boolean satisfiability to checking consistency of a single history reusing the encoding from Furbach et al. [18]. Let $\phi$ 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 $\phi$ 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, $p_{false}^i$ and $p_{true}^i$. Site $p_{false}^i$ first writes each $C_i \in Pos(x_i)$ (in the order they appear in $C_1, \ldots, C_k$) to variable $y$, and then, it writes $i$. Similarly, Site $p_{true}^i$ writes each $C_i \in Neg(x_i)$ (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 $rd(y) \triangleright 1\cdot rd(y) \triangleright n$ followed by $rd(y) \triangleright C_1 \cdots rd(y) \triangleright C_k$.

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

(⇐) This direction follows from the proof of [18]. They show that if $\phi$ is satisfiable, the history $h$ is sequentially consistent.

(⇒) Assume $h$ is $CC$. Then, there exists $co$, such that, for all $o \in O$, there exists $p_o \in S_{eq}$, such that $AxCausal$ and $AxCausalValue$ hold. In particular, each $rd(y) \triangleright i$ of $p_{eval}$ must have $wr(y, i)$ in its causal past.

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

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

Moreover, each read $rd(y) \triangleright C_i$ has a corresponding $wr(y, C_i)$, meaning that $\phi$ is satisfiable. $\square$

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 $Pld \times M \times D \times D$, meaning that $m$ was called with argument value $arg$ on site $p$ and returned value $rv$. An implementation $I$ is a set of executions which is prefix-closed (if $I$ contains an execution $e \cdot e'$, $I$ 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 $h = (O, PO, \ell)$ contains the same operations as $e$, and orders $o_1 <_P o_2$ 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 $I$ 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 $S_{eq}$, 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 \( I \) is not causally consistent (resp., \( \mathsf{CC}, \mathsf{CM}, \mathsf{CCr} \)).

**Definition 1.** Let \( \Sigma_{\mathsf{PCP}} \) be a finite alphabet. PCP asks, given \( n \) pairs \((u_1, v_1), \ldots , (u_n, v_n) \in (\Sigma_{\mathsf{PCP}} \times \Sigma_{\mathsf{PCP}})^*\), whether there exist \( t_1, \ldots , t_k \in \{1, \ldots , n\} \) such that \( u_{t_1} \cdots u_{t_k} = v_{t_1} \cdots v_{t_k} \), with \( (k > 0) \).

From a high-level view, both proofs operate similarly. We build, from a PCP instance \( P \), an implementation \( I \) (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 \( I \) contains an execution which is not causally consistent (resp., \( \mathsf{CC}, \mathsf{CM}, \mathsf{CCr} \)) with respect to \( S \) (with respect to a bounded version of \( S_{\mathsf{ROW}} \) for the second proof).

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

**Definition 2.** Two sequences \((u, v)\) in \( \Sigma_{\mathsf{PCP}}^* \) form a valid answer if \( u = v \) and they can be decomposed into \( u = u_1 \cdot u_2 \cdots u_k \) and \( v = v_1 \cdot v_2 \cdots v_k \), with each \((u_i, v_i)\) corresponding to a pair of problem \( P \).

Therefore, \( I \) is not causally consistent, if and only if \( I \) 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., \( \mathsf{CC}, \mathsf{CM}, \mathsf{CCr} \)), 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 \( u \parallel v \subseteq \Sigma^* \) inductively: \( \varepsilon \parallel v = \{v\} \), \( u \parallel \varepsilon = \{u\} \) and \( (a \cdot u)(b \cdot v) = a \cdot (u \parallel (b \cdot v)) \cup b \cdot (a \cdot u) \parallel v \), with \( a, b \in \Sigma \).

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

**Lemma 4.** The shuffling problem is undecidable.

We now give the undecidability theorem for causal consistency (resp., \( \mathsf{CC}, \mathsf{CM}, \mathsf{CCr} \)) by reducing the shuffling problem to the problem of verifying (non-)causality. The idea is to let one site simulate words from \( \Sigma_u^* \), and the second site from \( \Sigma_v^* \). 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 \( u \parallel v \) belongs to \( L \), i.e. \( u \parallel v \cap L = \emptyset \).

**Theorem 1.** Given an implementation \( I \) and a specification \( S \) given as regular languages, checking whether all executions of \( I \) are causally consistent (resp., \( \mathsf{CC}, \mathsf{CM}, \mathsf{CCr} \)) 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 imple-mentation is not \( \mathsf{CC} \) (resp., \( \mathsf{CM}, \mathsf{CCr} \)) 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 \( S_{\mathsf{ROW}} \)) and we must encode them using appropriately placed read and write operations.

**Theorem 2.** Given an implementation \( I \) as a regular language, checking whether all executions of \( I \) are causally consistent (resp., \( \mathsf{CC}, \mathsf{CM} \), \( \mathsf{CCr} \)) with respect to \( S_{\mathsf{ROW}} \) 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 \( \mathsf{CC} \), we provide in Section 7.2 four bad patterns such that, a differentiated history \( h \) is \( \mathsf{CC} \) if and only if \( h \) contains none of these bad patterns. We give similar lemmas for \( \mathsf{CM} \) and \( \mathsf{CCr} \).

#### 7.1 Differentiated Histories

Formally, a history \((O, PO, \ell)\) is said to be differentiated if for all \( o_1 \neq o_2 \), if \( \ell(o_1) = wr(x) \triangleright d_1 \) and \( \ell(o_2) = wr(x) \triangleright d_2 \), then \( d_1 \neq d_2 \), and there are no operation \( wr(x, 0) \) (which writes the initial value). Let \( H \) be a set of labeled posets. We denote by \( H_e \) 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., \( \mathsf{CC}, \mathsf{CM}, \mathsf{CCr} \)) with respect to the read/write memory if and only if \( H_e \) is causally consistent (resp., \( \mathsf{CC}, \mathsf{CM}, \mathsf{CCr} \)) with respect to the read/write memory.

#### 7.2 Characterizing Causal Consistency (CC)

Let \( h = (O, PO, \ell) \) be a differentiated history. We now define and explain the bad patterns of \( \mathsf{CC} \). 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:

\[
(\ell(o_1, o_2) \mid \exists x \in X, d \in D, \ell(o_1) = wr(x, d) \land \ell(o_2) = rd(x) \triangleright d).
\]

The relation \( CO \) is defined as \( CO = (PO \cup RF)^+ \).
Table 2: All bad patterns defined in the paper.

<table>
<thead>
<tr>
<th>CC</th>
<th>CM</th>
<th>CCv</th>
</tr>
</thead>
<tbody>
<tr>
<td>CyclicCO</td>
<td>CyclicCO</td>
<td>CyclicCO</td>
</tr>
<tr>
<td>WriteCOInitRead</td>
<td>WriteCOInitRead</td>
<td>WriteCOInitRead</td>
</tr>
<tr>
<td>ThinAirRead</td>
<td>ThinAirRead</td>
<td>ThinAirRead</td>
</tr>
<tr>
<td>WriteCORead</td>
<td>WriteCORead</td>
<td>WriteCORead</td>
</tr>
<tr>
<td>WriteHBlInitRead</td>
<td>WriteHBlInitRead</td>
<td>WriteHBlInitRead</td>
</tr>
<tr>
<td>CyclicHB</td>
<td>CyclicCF</td>
<td>CyclicCF</td>
</tr>
</tbody>
</table>

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 (PO \cup 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, \( \text{wr}(x,1) \) is causally related (through relation CO) to \( \text{wr}(x,2) \), which is causally related to \( \text{rd}(x) \). Intuitively, this means that the site executing \( \text{rd}(x) \) is aware of both writes \( \text{wr}(x,1) \) and \( \text{wr}(x,2) \), but chose to order \( \text{wr}(x,2) \) before \( \text{wr}(x,1) \), while \( \text{wr}(x,1) \) is causally related to \( \text{wr}(x,2) \). 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 \( S_{\text{RW}} \) if and only if h does not contain one of the following bad patterns: CyclicCO, WriteCOInitRead, ThinAirRead, WriteCORead.

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

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

First, we show that CO \( \subseteq \text{co} \). Given the specification of rd’s, and given that h is differentiated, we must have RF \( \subseteq \text{co} \). Moreover, by axiom A\text{Causal}, PO \( \subseteq \text{co} \). Since co is a transitive order, we thus have (PO \cup RF) \* \( \subseteq \text{co} \) and CO \( \subseteq \text{co} \).

(WriteCOInitRead) If there is a rd(x) \( \triangleright 0 \) operation \( r \) and an operation \( w \) such that \( w <_{\text{CO}} r \) and \( \text{var}(w) = \text{var}(r) \), then there is a cycle in \( \text{PO} \cup \text{RF} \) (in CO)

ThinAirRead there is a rd(x) \( \triangleright v \) operation \( r \) such that \( v \neq 0 \), and there is no w operation with \( w <_{\text{RF}} r \).

WriteCORead there exist write operations \( w_1, w_2 \) and a read operation \( r_1 \) in O such that \( w_1 <_{\text{CO}} w_2 <_{\text{RF}} r_1 \), and \( \text{var}(w_1) = \text{var}(w_2) \).

WriteHBlInitRead there is a rd(x) \( \triangleright 0 \) operation \( r \), and an operation \( w \) such that \( w <_{\text{HB}} x \) and \( \text{var}(w) = \text{var}(r) \), for some \( o \), with \( r <_{\text{PO}} o \).

Cyclichb there is a cycle in HB, for some \( o \in O \).

CyclicCF there is a cycle in CF \cup CO.
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 \(w_1 \not\subset_{CF} w_2\) 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 \(wr(x,1)\) operation \(w_1\) is causally related to the \(rd(x) \triangleright 2\) operation, so we have \(w_1 \not\subset_{CF} w_2\), where \(w_2\) is the \(wr(x,2)\) operation. Symmetrically, \(w_2 \not\subset_{CF} w_1\), 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 \(wr(x,1)\) and \(wr(x,2)\).

The formal definition of the conflict relation is the following.

Definition 5. We define the conflict relation \(\subset_{CF} \subseteq O \times O\) to be the smallest relation such that: for all \(x \in X\), \(d_1 \neq d_2 \in \mathbb{N}\), and operations \(w_1, w_2, r_2\), if
\[
\begin{align*}
&\cdot \ w_1 \not\subset_{CF} r_2, \\
&\cdot \ r_2 \not\subset_{PO} o, \\
&\cdot \ r_2 \not\subset_{PO} w_2, \\
&\cdot \ w_2 \not\subset_{CF} w_1, \\
&\cdot \ (r_2, w_1) \not\subset_{CF} (w_2, w_1), \\
&\cdot \ (r_2, w_1) \not\subset_{CF} (w_2, w_2), \\
&\cdot \ (r_1, w_1) \not\subset_{CF} (w_2, w_2),
\end{align*}
\]
then \(w_1 \not\subset_{CF} w_2\).

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 \(wr(x,1) \not\subset_{PO} wr(x,1) \subset_{HB} wr(x,2) \subset_{PO} rd(z) \triangleright 0\), where \(r_2\) is the \(rd(x) \triangleright 2\) operation. The edge \(wr(x,1) \subset_{HB} wr(x,2)\) is induced by the fact that \(wr(x,1) \subset_{CO} rd(x) \triangleright 2\).

The formal definition of HB is the following.

Definition 6. Given \(o \in O\), we define the happened-before relation for \(o\), noted \(\subset_{HB}\), to be the smallest relation such that:
\[
\begin{align*}
&\cdot \ CO_{CausalPast(o)} \subseteq \subset_{HB}, \text{ and} \\
&\cdot \ \subset_{HB} \text{ is transitive, and} \\
&\cdot \ \text{for } x \in X, \text{ and } d_1 \neq d_2 \in \mathbb{N}, \text{ if} \\
&\quad \cdot \ w_1 \subset_{HB} r_2, \\
&\quad \cdot \ r_2 \not\subset_{PO} o, \\
&\quad \cdot \ r_2 \not\subset_{PO} o, \\
&\quad \cdot \ (w_1) \subset_{HB} (w_2), \\
&\quad \cdot \ (w_1) \subset_{HB} (w_2), \\
&\quad \cdot \ (w_2) \subset_{HB} (w_2), \\
&\quad \cdot \ (w_2) \subset_{HB} (w_2),
\end{align*}
\]
then \(w_1 \subset_{HB} w_2\).

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 \(r_2 \not\subset_{PO} o\), 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 \(h = (O, PO, ℓ)\) be a differentiated history. Computing the relations RF, CO, CF, and HB, 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, (for some \(o \in O\)). The same holds for the other relations. The relation HB 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 can consist in adding an edge by transitivity, which takes \(O(n^3)\) time.

Thus the whole computation of HB 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 \(h = (O, PO, ℓ)\) 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 RF, CO, CF, HB, (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 \(O(n^5)\).

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 \(M_{CC}\) that looks for the bad patterns leading to non-CC. More precisely, our goal is to define \(M_{CC}\) as a register automaton such that
(by an abuse of notation, the set of executions recognized by $M_{CC}$
is also denoted $M_{CC}$):

$$I \text{ is CC with respect to } S_{RW} \iff I \cap M_{CC} = \emptyset$$

where $I$ 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 $M_{CC}$, 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. Each state has a self-loop with any symbol containing value 5, which we do not represent. Two labels $p, m(\text{arg}) \triangleright \text{rw}$ above a transition denote two different transitions.](image)

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

9.1 Register Automata

Register automata [10] 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 $p, wr(x, 1)$ above the transition going from $q_1$ in Figure 4 is a form of pattern matching. If the automaton reads a tuple $(p_0, wr(x_0, 1))$, for some $p_0 \in \text{PId}, x_0 \in X$, 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 $\text{reg}_x \coloneqq x$, on this same transition, is used to store the value $x_0$ which was bound by $x$ in register $\text{reg}_x$. This ensures that the operations $\text{wr}(x, 2)$ and $\text{rd}(x) \triangleright 1$ that come later use the same variable $x_0$, thanks to the equality check $\text{reg}_x \equiv x$.

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

$M_{CC}$ (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 $w_1 <_{CO} w_2 <_{CO} r_1, w_1 <_{RF} r_1$, and $\text{var}(w_1) = \text{var}(w_2)$. 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 rd$(x) \triangleright 0$.

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

The registers of $M_{CC}$ 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, $M_{CC}$ 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 $M_{CC}$ is not CC. Reciprocally, we prove that for any differentiated execution of an implementation $I$ which is not CC, we can rename the values to obtain an execution with 5 values recognized by $M_{CC}$. By data independence of $I$, the renamed execution is still an execution of $I$.

Remark 2. Note here that the observer $M_{CC}$ 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 $I$ be a data independent implementation. $I$ is CC with respect to $S_{RW}$ if and only if $I \cap M_{CC} = \emptyset$.

This result allows to reuse any tool or technique that can solve reachability (in the system composed of the implementation and the observer $M_{CC}$) 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 $C$ for which reachability is decidable, making CC decidable (EXPSPACE-complete). We consider implementations $I$ 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 $I$ 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
implementation of $C$ 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 $M_{CC}$ we constructed only needs 5 values to detect all $CC$ violations. For this reason, when modeling an implementation in $C$, 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 $C$ can be modelled by a Vector Addition System with States (VASS) [24, 26], or a Petri Net [15, 36]. 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.

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 $M_{CC}$, and obtain a (normal) finite automaton. We then need to solve control-state reachability in the system composed of the VASS and the observer $M_{CC}$ 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 $C$ is EXPSPACE-complete, equivalent to control-state reachability in VASS [4]. Intuitively, a VASS can be modelled by an implementation in $C$, by using the unbounded unordered channels to simulate the counters of the VASS. (Similar to the reduction from $C$ to VASS outlined above, but in the opposite direction.) (2) Checking reachability can be reduced to verifying $CC$. Given an implementation $I$ in $C$, and a state $q$, knowing whether $q$ is reachable can be reduced to checking whether a new implementation $I'$ is not causally consistent. $I'$ is an implementation which simulates $I$, 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 $I$ be a data independent implementation in $C$. Verifying $CC$ of $I$ with respect to the read/write memory is EXPSPACE-complete (in the size of the VASS of $I$).

### 11. Related Work

We et al. [41] 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. [18] showed that checking causal consistency (see Definition 1) 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 [23] or eventual consistency [39]. In particular, it was shown that checking linearizability is an EXPSPACE-complete problem when the number of sites is bounded [3, 21]. 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, 22]. 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. [31] 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, Cs, 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

A. Differentiated Histories

We detail here the results of Section 7.1. We identify here conditions under which it is enough to check the causal consistency of only a subset $H' \subseteq H$ of histories, while ensuring that all histories $H$ are causally consistent.

We then use this notion to prove that it is enough to check causal consistency with respect to the read/write memory for histories which use distinct $\varphi$ values.

A.1 Reduction

Let $\mathcal{R}$ be a relation over labeled posets. A subset $H' \subseteq H$ is said to be complete for $H$ if: for all $h \in H$, there exists $h' \in H'$, $h' \prec_R h$.

When checking whether a set of histories $H_1$ is a subset of a set $H_2$ which is upward-closed with respect to $\mathcal{R}$, it is sufficient to check the inclusion for a complete subset $H'_1 \subseteq H_1$.

**Lemma 10** (Complete Sets of Histories). Let $\mathcal{R}$ be a relation, and let $H_1 \subseteq H_2$ be a complete set of labeled posets using relation $\mathcal{R}$. Let $H_2$ be a set of histories which is upward-closed with respect to $\mathcal{R}$, we have $H_1 \subseteq H_2$ if and only if $H'_1 \subseteq H'_2$.

**Proof.** ($\Rightarrow$) Holds because $H'_1$ is a subset of $H_1$.

($\Leftarrow$) Assume $H'_1 \subseteq H_2$ and let $h_1 \in H_1$. Since $H'_1$ is complete for $H_1$, we know there exists $h'_1 \in H'_1$ such that $h'_1 \prec_R h_1$. By assumption, $h'_1 \in H_2$. Finally, since $H_2$ is upward-closed, $h_1 \in H_2$.

\[\square\]

Given a function $f : \mathcal{D} \to \mathcal{D}$ and a tuple $a \in \mathcal{M} \times \mathcal{D}$ or $a \in \mathcal{M} \times \mathcal{D} \times \mathcal{D}$, we denote by $a[f]$ the tuple where each occurrence of $d \in \mathcal{D}$ has been replaced by $f(d)$. We lift the notation to $\mathcal{M} \times \mathcal{D}$ and $\mathcal{M} \times \mathcal{D} \times \mathcal{D}$ labeled posets by changing the labels of the elements.

We lift the notation to sets of labeled posets in a point-wise manner.

Let $F \subseteq \mathcal{D} \to \mathcal{D}$ be a set of functions.

**Definition 7.** $S$ is $F$-invariant if for all $f \in F$, $S[f] \subseteq S$.

We define a relation $\overset{F}{\Rightarrow}$ as follows: $h_1 \overset{F}{\Rightarrow} h_2 \iff \exists f \in F. h_2 = h_1[f]$. Let $S$ be a specification. We denote by $\mathcal{CC}(S)$ (resp., $\mathcal{CM}(S)$, $\mathcal{CCv}(S)$) the set of histories which are $\mathcal{CC}$ (resp., $\mathcal{CM}$, $\mathcal{CCv}$) with respect to $S$. We show that for any specification which is $F$-invariant, the set $\mathcal{CC}(S)$ (resp., $\mathcal{CM}(S)$, $\mathcal{CCv}(S)$) is upward-closed with respect to the relation $\overset{F}{\Rightarrow}$.

**Lemma 11.** Let $F \subseteq \mathcal{D} \to \mathcal{D}$ be a set of functions. Let $S$ be a specification which is $F$-invariant. The set $\mathcal{CC}(S)$ (resp., $\mathcal{CM}(S)$, $\mathcal{CCv}(S)$) is upward-closed with respect to $\overset{F}{\Rightarrow}$.

**Proof.** We show the proof for $\mathcal{CC}(S)$, but the proof can be directly adapted to the sets $\mathcal{CM}(S)$ and $\mathcal{CCv}(S)$. Let $h = (O, <, \ell) \in \mathcal{CC}(S)$ and $h'$ such that $h \overset{F}{\Rightarrow} h'$. We know there exists $f \in F$ such that $h' = h[f]$. As a result, $h$ and $h'$ have the same underlying poset $(O, <)$. Let $h' = (O, <, \ell')$.

Let $o \in O$. By axiom $\text{AxCausalValue}$, we know there is $\rho_o \in S$ such that $\text{CausalHist}(o)(\ell) \leq \rho_o$, where $\text{CausalHist}(o) = (\text{CausalPast}(o), o, \ell)$. We have $(\text{CausalPast}(o), o, \ell') = \text{CausalHist}(o)[f]$. Therefore, by defining $\rho'_o = \rho_o[f]$, we have $(\text{CausalPast}(o), o, \ell') \leq \rho'_o$. Since $S$ is $F$-invariant, $\rho'_o \in S$, and axiom $\text{AxCausalValue}$ holds for history $h'$.

We conclude that $h'$ is in $\mathcal{CC}(S)$.

\[\square\]
Corollary 1. Let $F \subseteq \mathcal{D} \rightarrow \mathcal{D}$ be a set of functions. Let $H$ be a set of histories, and $H' \subseteq H$ a complete set of histories, using relation $F$. Let $S$ be specification which is $F$-invariant. Then, $H$ is $CC$ (resp., $CM$, $CCx$) with respect to $S$ if and only if $H' \subseteq CC(S)$ of histories, and $H' \subseteq H$ a complete set of histories, using relation $F$. Let $S$ be specification which is $F$-invariant. Then, $H$ is $CC$ (resp., $CM$, $CCx$) with respect to $S$.

Proof. Using Lemmas 10 and 11, we know $H' \subseteq CC(S)$ if and only if $H \subseteq CC(S)$. The same applies for the sets $CM(S)$ and $CCx(S)$.

A.2 Data Independence

We show how to apply the notion of completeness for the verification of the read/write memory. Most implementations used in practice are 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 show it is enough to verify causal consistency for histories which do not write twice the same value on the same variable, and which never writes the initial value 0, called differentiated histories.

Formally, a history $(O, PO, \ell)$ is said to be differentiated if:
- For all $o_1 \neq o_2, x \in X, \ell(o_1) = wr(x, d_1)$ and $\ell(o_2) = wr(x, d_2)$ and implies that $d_1 \neq d_2$, and
- For all $x \in X, h$ does not contain $wr(0, 0)$ operation.

Let $H$ be a set of labeled posets. We denote by $H_s$ the subset of differentiated histories of $H$.

A data-renaming function from $\mathcal{D}$ to $\mathcal{D}$ which modifies the data values of operations. More precisely, we can build a data-renaming $f$ from any function $f_0 : N \rightarrow N$ in the following way.

Remember that for the read/write memory, $\mathcal{D}$ is the set $\mathcal{X} \times N \subseteq \mathcal{X} \cup N \cup \{1\}$, and we define:
- $f(x, arg) = (x, f_0(arg))$ for $x \in \mathcal{X}$, $arg \in N$,
- $f(x) = x$ for $x \in \mathcal{X}$,
- $f(rv) = f_0(rv)$ for $rv \in N$,
- $f(1) = 1$.

Let $F_{Data}$ be the set of all data-renamings.

Definition 8. A set of histories is data independent if the subset $H_s \subseteq H$ is complete using relation $F_{Data}$, and $H$ is $F_{Data}$-invariant.

Remark 3. This definition corresponds to the other definition of data independence we have in Section 7.1.

Using the following lemma and then applying Corollary 1, we obtain that it is enough to check causal consistency for histories which are differentiated (to ensure the causal consistency of all histories).

Lemma 12. The $S_{RW}$ specification is $F_{Data}$-invariant.

Proof. Let $f \in F_{Data}$ and let $\rho \in S_{RW}$. We can see that $\rho[f] \in S_{RW}$, as changing the written and read values in a valid sequence of $S_{RW}$ yields a valid sequence of $S_{RW}$.

Lemma 5. Let $H$ be a data independent set of histories. Then, $H$ is causally consistent (resp., $CC$, $CM$, $CCx$) with respect to the read/write memory if and only if $H_s$ is causally consistent (resp., $CC$, $CM$, $CCx$) with respect to the read/write memory.

Proof. Since $H$ is data independent, we have that $H_s \subseteq H$ is complete using relation $F_{Data}$. The result then follows directly from Corollary 1 and Lemma 12.

B. Undecidability of Causal Consistency For 2 Sites

Lemma 4. The shuffling problem is undecidable.

Proof. Let $\Sigma_{PCP} = \{a, b\}$ and $(u_1, v_1, \ldots, u_n, v_n) \in (\Sigma_{PCP} \times \Sigma_{PCP})$ be $n$ pairs forming the input of a PCP problem. Let $\Gamma_u = \{a_v, b_v\}$ and $\Gamma_v = \{a_v, b_v\}$ be two disjoint copies of $\Sigma_{PCP}$, and let $h : (\Gamma_u \cup \Gamma_v) \rightarrow \Sigma_{PCP}$, $h_u : \Sigma_{PCP} \rightarrow \Gamma_u$, $h_v : \Sigma_{PCP} \rightarrow \Gamma_v$, be the homomorphisms which map corresponding letters. Moreover, let $s_u$ and $s_v$ be two new letters. Let $\Sigma_u = \{a_v, b_v, s_u\}$, $\Sigma_v = \{a_v, b_v, s_v\}$, and $\Sigma = \Sigma_u \cup \Sigma_v$.

Our goal is to define a regular language $L \subseteq \Sigma^*$ such that:

The idea is to encode in $u$ the sequence $u_1, \ldots, u_n$ as $h_u(u_1)$, $s_u, \ldots, h_u(u_n)$ by using $s_u$ as a separator (and end marker), and likewise for $v$ with the separator $s_v$. Then, we define the language $L$ by a disjunction of regular properties that no shuffling of an encoding of a valid PCP answer to $P$ could satisfy.

Formally, $w \in L$ if one of the following conditions holds:
1. when ignoring the letters $s_u$ and $s_v$, $w$ starts with an alternation of $\Gamma_u$ and $\Gamma_v$ such that two letters do not match $w_{\Gamma_u \Gamma_v} = (\Gamma_u, \Gamma_v)^* (a_v, b_v + b_v, a_v) \Sigma^*$
2. when ignoring the letters $s_u$ and $s_v$, $w$ starts with an alternation of $\Gamma_u$ and $\Gamma_v$ and ends with only $\Gamma_u$ letters or only $\Gamma_v$ letters $w_{\Gamma_u \Gamma_v} = (\Gamma_u, \Gamma_v)^* (\Gamma_u + \Gamma_v)^*$
3. when only keeping $s_u$ and $s_v$ letters, either $w$ starts with an alternation of $s_u$ and $s_v$ and ends with only $s_u$ or only $s_v$, or $w$ is the empty word $\varepsilon$.
4. $w$ contains a letter from $\Gamma_u$ not followed by $s_u$, or a letter from $\Gamma_v$ not followed by $s_v$.
5. $w$ starts with an alternation of $\Gamma_u^* s_u$ and $\Gamma_v^* s_v$ such that one pair of $\Gamma_u^*, \Gamma_v^*$ is not a pair of our PCP instance.

We can now show that equivalence (1) holds.

Let $k > 0, i_1, \ldots, i_k \in \{1, \ldots, n\}$ such that $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$.

Let $u = h_u(u_{i_1}) \ldots h_u(u_{i_k}) \cdot s_u \ldots h_u(u_n)$. Let $v = h_v(v_{i_1}) \ldots h_v(v_{i_k}) \cdot s_v \ldots h_v(v_n)$. We want to show that no word $w$ in the shuffling of $u$ and $v$ satisfies one of the conditions of $S$. If $w$ starts with an alternation of $\Gamma_u^* s_u$ and $\Gamma_v^* s_v$, then any pair of letters match, and thus, condition 1 cannot hold. Condition 2 cannot hold since $w$ contains as many letters from $\Gamma_u$ as from $\Gamma_v$. Likewise for condition 3 since $w$ contains as many $s_u$ as $s_v$ (and at least 1).

Since $u$ (resp., $v$) does not contain a letter from $\Gamma_u$ not followed by $s_u$ (resp., a letter from $\Gamma_v$ not followed by $s_v$), neither does $w$, which shows that condition 4 does not hold. Finally, if $w$ starts with an alternation of $\Gamma_u^* s_u$ and $\Gamma_v^* s_v$, then all the corresponding pairs of $\Gamma_u^*, \Gamma_v^*$ are pairs from the PCP input, and condition 5 cannot hold either.

Let $w \in \Sigma_{\Gamma_u}^* w \in \Sigma_{\Gamma_v}^*$ such that $w \in \Gamma_u \Gamma_v = \varepsilon$. Since no word in $w \Gamma_u \Gamma_v$ satisfies condition 3, nor condition 4, $u$ ends with $s_u$, $v$ ends with $s_v$, and $w$ has as many $s_u$ as $s_v$ (and at least 1). This shows that, $u = x_1 \ldots x_k s_u$ and $v = y_1 \ldots y_k s_v$ for some $k > 0, x_1, \ldots, x_k \in \Gamma_u, y_1, \ldots, y_k \in \Gamma_v$. Moreover, since no word in $u \Gamma_v \Gamma_u$ satisfies condition 5, for any $j, (x_j, y_j)$ corresponds to a pair $(u_{i_j}, v_{i_j})$ of our input $P$ – more precisely, $h(x_j) = u_{i_j}$ and $h(y_j) = v_{i_j}$ for some $i_j \in \{1, \ldots, n\}$. Finally, the fact that no word in $u \Gamma_v \Gamma_u$ satisfies condition 1, nor condition 2 ensures that $h(x_1 \ldots x_k) = h(y_1 \ldots y_k)$ and that the PCP problem $P$ has a positive answer $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$.
Theorem 1. Given an implementation $I$ and a specification $S$ given as regular languages, checking whether all executions of $I$ are causally consistent (resp., $CC$, $CM$, $CCv$) with respect to $S$ is undecidable.

Proof. Let $L$ be a regular language over $\Sigma = \Sigma_u \cup \Sigma_v$. We construct an implementation $I$ and a specification $S$, in order to reduce the shuffling problem (undecidable by Lemma 4) to the negation of causal consistency.

Said differently, if (and only if) the shuffling problem has a positive answer, i.e. there exist $u \in \Sigma_u^*$ and $v \in \Sigma_v^*$ such that $u|v \cap L = \emptyset$, then there exists an execution in $I$ which is not causally consistent (resp., $CC$, $CM$, $CCv$) with respect to $S$.

For the methods, we use $M = \{m_a \mid a \in \Sigma_u\} \cup \{m_b \mid b \in \Sigma_v\} \cup \{End_a, End_b\}$, and $D = \{F, T\}$ for the domain of the return values. The return values are only relevant for the method $End_b$, so we do not represent return values for the other methods. We assume here that methods do not take arguments, as we do not need them for the reduction. As a result, a specification is a set of sequences labeled by $M \times D$ (method, return value).

Let $u \in \Sigma_u$ and $v \in \Sigma_v$. $I$ produces, for any such pair, an execution whose history is $H_{(u, v)}$, described hereafter. The specification $S$ is then built in such a way that $H_{(u, v)}$ is not causally consistent if and only if $u|v \cap L = \emptyset$. Therefore, the shuffling problem has a positive answer if and only if $I$ contains an execution which is not causally consistent.

Here is the description of an execution corresponding to a pair $u \in \Sigma_u$ and $v \in \Sigma_v$, whose history is $H_{(u, v)}$. The implementation contains two sites, $p_A$ and $p_B$. Site $p_A$ execute $m_a$ operations for the letters $a$ of $u$. Site $p_B$ execute $m_b$ operations for the letters $b$ of $v$. Method $End_a$ is then executed on site $p_A$. Site $p_A$ then sends a message to $p_B$, informing $p_B$ that a method $End_b$ returning $T$ can now be executed on $p_B$. When the message is received by $p_B$, a method $End_b$ returning $T$ is then executed on $p_B$. (If method $End_B$ is called on site $p_B$ another time, $p_B$ returns $F$).

Remark 4. If a method $m_b$ with $b \in \Sigma_v$ gets executed on site $p_A$, then $p_A$ does not send the message to site $p_B$. If a method $m_a$ with $a \in \Sigma_u$ gets executed on site $p_B$, then $p_B$ return $F$ when method $End_b$ gets called.

Such executions will be causally consistent by default, by construction of $S$, defined below, because they will not contain any $End_b \triangleright T$ operations. (The specification $S$ contains, among other sequences, any sequence which does not contain $End_b \triangleright T$ operations.)

Formally, the set of executions $I$ is a regular language, and can be represented by the finite automaton given in Figure 6 (we do not represent the executions described in Remark 4, which can never lead to non-causally consistent executions).

The epsilon transition going from $q_2$ to $q_3$ represents the fact that site $p_B$ receives the message sent by $p_A$. After this, site $p_B$ can execute a $End_b$ method returning $T$. This epsilon transition is only here for clarity and can be removed.

We here do not represent transitions with $End_b$ returning $F$, as $End_b \triangleright F$ operations are ignored by the specification $S$. They can be added as self-loops to the automaton.

The specification $S$ is defined to contain any word $w$ such that:

- $w$ does not contain $End_b \triangleright T$, or
- when ignoring $End_b \triangleright F$, $w$ is of the form $L' \triangleright End_a \triangleright End_b \triangleright T$, where $L'$ is $L$ with every letter $a \in \Sigma_u$ replaced by $m_a$ and every letter $b \in \Sigma_v$ replaced by $m_b$.

We now prove the following equivalence:

1. $I$ is not $CC$ (resp., $CM$, $CCv$) with respect to $S$,
2. $\exists u \in \Sigma_u^*, v \in \Sigma_v^*, u|v \cap L = \emptyset$,

Figure 6: Finite automaton describing the executions of implementation $I$ of Theorem 1. All states are accepting.

(2) $\Rightarrow$ (1) Let $u \in \Sigma_u^*$ and $v \in \Sigma_v^*$ such that $u|v \cap L = \emptyset$. We construct an execution $e$ in $I$ which is not causally consistent (resp., $CC$, $CM$, $CCv$). The execution $e$ follows the description above, and the history of $e$ is $H_{(u, v)}$.

Site $p_B$ executes the sequence of operations $m_a$ for each letter $b$ of $v$. Independently, site $p_A$ executes the sequence of operations $m_b$ for each letter $a$ of $u$. The site $p_A$ then executes a $End_a$ operation and sends a message to site $p_B$. After $p_B$ receives the message, $p_B$ executes a $End_b \triangleright T$ operation.

Assume by contradiction that $e$ is $CC$ (a contradiction here also proves that $e$ cannot be $CM$ nor $CCv$). There must thus exists a causal order $o$ (containing the program order $PO$). Let $o$ be the $End_b \triangleright T$ operation of $e$ We know there exists $\rho_o \in S$, such that $CausalHist(o)(a) \leq \rho_o$.

Since $\rho_o$ contains $o$, by definition of $S$, $\rho_o$ must be of the form $\rho_o': End_a \triangleright End_b \triangleright T$, with $\rho_o'$. In particular, this means that $\rho_o$ must contain the $End_a$ operation of $e$. By transitivity of $\triangleright$, and because $PO \subseteq o$, $\rho_o$ must contain all operations of $e$.

Thus, the sequence $\rho_o'$ effectively defines a shuffling of $u$ and $v$ which is in $L$, contradicting the assumption that $u|v \cap L = \emptyset$.

(1) $\Rightarrow$ (2) Let $e$ be an execution of $I$ which is not causally consistent (resp., $CC$, $CM$, $CCv$). It must contain a $End_b \triangleright T$ operation, otherwise, by definition of $S$, $e$ is causally consistent regardless of how we define the causality order $co$ (as a strict partial order).

Note that there can only be one $Check \triangleright T$ operation. Indeed, after executing $End_b \triangleright T$, site $p_B$ only returns $F$ when method $End_b$ gets called.

Also, for $p_B$ to execute a $End_b \triangleright T$ operation, it must be the case that $p_A$ executed a $End_a$ operation $o$, and sent a message to $p_B$.

This means that, prior to $o$, $p_A$ must have executed a sequence of $m_a$ operations, with $a \in \Sigma_u$, corresponding to a word in $u \in \Sigma_u$.

Similarly, prior to executing the $End_b \triangleright T$ operation $p_B$ must have executed a sequence of $m_b$ operations corresponding to a word in $v \in \Sigma_v$.

Assume by contradiction that there exists a word $w$ in the shuffling of $u$ and $v$ which belongs to $L$, i.e. assume by contradiction that $u|v \cap L \neq \emptyset$. Using this, we can construct a sequence $\rho_o \in S$, containing the $End_a \triangleright T$ and $End_b \triangleright T$ operations as well as all operations corresponding to $u$ and $v$, to form a sequence which belongs $S$. This means that $e$ must be causally consistent (for any definition) and we have a contradiction.
This ends the proof of equivalence between statements 1 and 2, and ends the reduction from the shuffling problem to checking whether an implementation is not causally consistent. This implies that checking whether a implementation is causally consistent is not decidable.

C. Undecidability for Non-Data-Independent Read/Write Memory Implementations

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

Proof. Let $\Sigma_{PCP} = \{a, b\}$ and $(u_1, v_1), \ldots, (u_n, v_n) \in (\Sigma_{PCP} \times \Sigma_{PCP})$ be $n$ pairs forming the input of a PCP problem $P$. We call these pairs dominoes.

Our goal is to build an implementation $I$ such that $I$ is not causally consistent (resp., CC, CM, CCR) with respect to the read/write memory if and only if the problem $P$ has a positive answer: $u_1 \ldots u_k = v_1 \ldots v_k$, $(k > 0)$.

Two sequences $(u, v)$ in $\Sigma_{PCP}$ form a sequence of dominoes if they can be decomposed into $u = u_{i_1} \ldots u_{i_k}$ and $v = v_{i_1} \ldots v_{i_k}$, with each $(u_{i_j}, v_{i_j})$ corresponding to a pair of $P$. They form a valid answer if we additionally have $u = v$. A valid answer corresponds to a positive answer for the PCP problem $P$.

Reduction Overview The implementation $I$ will produce, for each sequence of dominoes $(u, v)$, an execution whose history is $H_{(u,v)}$, defined thereafter.

We construct $H_{(u,v)}$ so that $H_{(u,v)}$ is not causally consistent (resp., CC, CM, CCR) if and only if $u = v$.

Therefore, if (and only if) $I$ is not causally consistent (resp., CC, CM, CCR), (and can produce a history which is not causally consistent), the PCP instance $P$ has a positive answer.

Construction of one History Given a letter $L \in \Sigma_{PCP}$, we define $L' = b$ if $L = a$, $L' = a$ if $L = b$.

Let $u = U_1 \ldots U_m$, and $v = V_1 \ldots V_n$, with $m, n > 0$, and $U_i, V_i \in \Sigma_{PCP}$ for all $i$. We depict in Figure 7 the history $H_{(u,v)}$ in $I$ corresponding to $(u, v)$. We show in Lemma 13 that $H_{(u,v)}$ is not causally consistent (for any definition) if and only if $u = v$.

To define $H_{(u,v)}$, we make use of the construct $\text{uniq}_{\text{rd}}(x) \triangleright d$, which denotes the sequence of operations $\text{wr}(x, 0) \cdot \text{rd}(x) \triangleright d \cdot \text{wr}(x, 0)$, for $x \in X$, and $d \neq 0$. This is only a notation, and does not imply that the three operations must be executed atomically. It is introduced only to simplify the presentation of the proof.

This construct ensures the useful property that $\text{uniq}_{\text{rd}}(x) \triangleright d$ operations made by the same site $p$ need distinct $\text{wr}(x, 1)$ to read from, as site $p$ overwrite $x$ with $\text{wr}(x, 0)$ after reading $\text{rd}(x) \triangleright d$. More generally, for any $m \in \mathbb{N}$, if a site $p$ does $m$ operations $\text{uniq}_{\text{rd}}(x) \triangleright d$ for some $d \neq 0$, then there need to be at least $m$ distinct $\text{wr}(x, d)$ in the execution for the execution to be causally consistent.

We now have all the ingredients needed to prove that the execution $H_{(u,v)}$ of Figure 7 satisfies the property we want: $H_{(u,v)}$ is causally consistent (resp., CC, CM, CCR) if and only if $u \neq v$. In the figure, we put the operations’ names between brackets, so that we can refer to them in the proof. When there is an operation name next to a $\text{uniq}_{\text{rd}}$ operation, it is actually the name corresponding to the underlying $\text{rd}$ operation.

The idea is the following. If $u$ and $v$ are different, then there exists $i$ such that $U_i \neq V_i$ and $U_i = V'_i$ (or $u$ and $v$ have different sizes). This means that the write $x_i$ can be used for the read $y_i$, which means that one write from the backup sites need not be used for $y_i$, and can be used instead for one of the three reads $r^a, r^b, r^#$. Moreover, two writes (out of three) from the extra sites can be used for the reads $r^a, r^b, r^#$, which makes the history $H_{(u,v)}$ causally consistent.

(The case where $u$ and $v$ have different sizes is handled thanks to the ticker sites, the symbol #, and the variables $S_u, S_v, M, N,$ and $O_h,$)

If $u$ and $v$ are equal, then $U_i$ is different than $V'_i$, for all $i$. Then, the writes of all backup processes must be used for the reads $y_i$ and
Figure 7: This history \( H_{(u,v)} \) corresponds to a sequence of dominoes \( (U_1 \ldots U_\mu; V_1 \ldots V_\nu) \), with \( U_i, V_i \in \{a, b\} \) for all \( i \). \( H_{(u,v)} \) is not causally consistent (resp., CC, CM, CCV) if and only (\( \mu = \nu \) and) \( U_1 \ldots U_\mu = V_1 \ldots V_\nu \), i.e. \( (U_1 \ldots U_\mu; V_1 \ldots V_\nu) \) form a valid answer for \( P \). This history uses 9 variables and a domain size of 4.
particular than each can be causally related to an operation \( r_f \), for \( i \in \{1, \ldots, \mu + 2\} \),
• \( t_i^r \to t_{j'}^r \), for \( i \in \{ \mu + 2, \mu + \nu + 2\} \),
• \( b_i \to y_{j'}, \) for \( i \in \{1, \ldots, B\} \), and where \( y_{j'} \) is the \( i \)th \( r_d(L) \) \( \triangleright b \) in \( p_v \).
• \( c_i \to z_{j'}, \) for \( i \in \{1, \ldots, C\} \), and where \( z_{j'} \) is the \( i \)th \( r_d(L) \) \( \triangleright \# \) in \( p_v \).
• \( a_i \to y_{j'}, \) for \( i \in \{1, \ldots, q - 1\} \), and where \( y_{j'} \) is the \( i \)th \( r_d(L) \) \( \triangleright a \) in \( p_v \).
• \( x_j \) to \( y_{j'} \).
• \( a_{i+1} \to y_{j'}, \) for \( i \in \{q + 1, \ldots, A\} \), and where \( y_{j'} \) is the \( i \)th \( r_d(L) \) \( \triangleright a \) in \( p_v \).
• \( a_A \) to \( r^b \),
• \( ch^n \) to \( r_{ch} \),
• \( e^{\times b} \) to \( r^{\#} \),
• \( e^{\times \#} \) to \( r^\# \),
• \( u_s \) to \( r_{f_v} \),
• \( s_u \) to \( r_{f_v} \),
• \( s_u \) to \( r_{f_v} \),
• \( ch^n \) to \( r_{ch} \).

By construction, \( co \) is a strict partial order, and we have \( PO \sqsubseteq co \). Also, for each read operation \( r \) (in particular \( r^b \) and \( r^\# \) ), we can construct a sequence, which respects the causality order, and containing all return values of the read operations before \( r \) in the program order. For every operation \( o \) of \( H_{(u,v)} \), there exists \( p_v \in S_{CO} \) such that \( CausalHist(o) \cap POPast(o) \) \( \subseteq p_v \) (axiom \( AxCausalSeq) \).

The key idea here is that, since \( U_j = V_{j'} \), \( y_j \) can read from \( x_j \). As a result, there is a \( wr(L, a) \) from the backup site \( a \) which is not needed for the \( uniq.r_d(L) \) \( \triangleright a \) operations of \( p_v \). We can thus use the last write from the backup site \( a \) (i.e. \( a_A \)), for the read \( r^a \). Then, we use \( ch^n \) to explain the return value of \( r_{ch} \), and we can therefore use \( e^{\times b} \) to explain the return value of \( r^b \), and \( e^{\times \#} \) to explain the return value of \( r^\# \).

Case \( \mu > \nu \). We prove that \( H_{(u,v)} \) is CM. We define the causality relation \( co \) as the transitive closure of the program order and the following constraints:

• \( t_i^r \to t_{j'}^r \), for \( i \in \{1, \ldots, \mu \} \),
• \( t_i^r \to t_{j'}^r \), for \( i \in \{1, \ldots, \mu \} \),
• \( b_i \to y_{j'}, \) for \( i \in \{1, \ldots, B\} \), and where \( y_{j'} \) is the \( i \)th \( r_d(L) \) \( \triangleright b \) in \( p_v \).
• \( c_i \to z_{j'}, \) for \( i \in \{1, \ldots, C\} \), and where \( z_{j'} \) is the \( i \)th \( r_d(L) \) \( \triangleright \# \) in \( p_v \).
• \( a_i \to y_{j'}, \) for \( i \in \{1, \ldots, q - 1\} \), and where \( y_{j'} \) is the \( i \)th \( r_d(L) \) \( \triangleright a \) in \( p_v \).
• \( x_j \) to \( y_{j'} \).
• \( a_{i+1} \to y_{j'}, \) for \( i \in \{q + 1, \ldots, A\} \), and where \( y_{j'} \) is the \( i \)th \( r_d(L) \) \( \triangleright a \) in \( p_v \).
• \( a_A \) to \( r^b \),
• \( ch^n \) to \( r_{ch} \),
• \( e^{\times b} \) to \( r^{\#} \),
• \( e^{\times \#} \) to \( r^\# \),
• \( u_s \) to \( r_{f_v} \),
• \( s_u \) to \( r_{f_v} \),
• \( s_u \) to \( r_{f_v} \),
• \( ch^n \) to \( r_{ch} \).

Case \( \nu > \mu \). Similar to the previous two cases. Here, \( x_{\mu+1} \) will be used for \( z_{\mu+1} \), thus allowing the write \( cc \) to be used for \( r^\# \).
Construction of the Implementation

We now describe how to build the implementation $\mathcal{I}$, such that $\mathcal{I}$ is not causally consistent with respect to the read/write memory if and only if the problem $P$ has a positive answer.

More precisely, we describe how to define $\mathcal{I}$ as a regular language, so that there are, for each sequence of dominoes $(u, v)$, an execution whose history is $H_{(u,v)}$.

In an execution of $\mathcal{I}$, the following happens: First, the extra sites, as well as the sites $p_{S_1}$ and $p_{S_2}$ execute their operations. Each ticker site executes its first operation. Then, site $p_0$ executes operations $r_{0h}$ and $d$.

After that, Site $p_0$ chooses non-deterministically a domino $(u_1, v_1)$ from the PCP instance $P$. It sends messages to the backup sites, the ticker sites, and $p_i$ so that they execute the operations corresponding to this domino $(u_1, v_1)$ (following Figure 7).

This step, of choosing non-deterministically a domino and what follows, can happen an arbitrary number of times.

All sites thus synchronize after each choice of a domino. The history of an execution $e$ of $\mathcal{I}$ thus always corresponds to a prefix of the history given in Figure 7.

Finally, the ticker sites, as well as $p_n$, $p_0$, and $p_f$ execute their last operations, as depicted in Figure 7.

Since the sites synchronize after each choice of a domino, $\mathcal{I}$ can be described by a regular language (or equivalently, by a distributed implementation where each site has a bounded local memory, and where the sites communicate through a network whose capacity is bounded).

Remark 5. In an implementation, each method can be called at any time, on any site. We handle this like in Theorem 1: if a site detects a method call that it is not expecting (i.e., that does not follow Figures 7), the implementation falls back to a default implementation which is causally consistent (resp., $CM$, $CM$, $CCr$).

Therefore, if $\mathcal{I}$ can produce an execution which is not causally consistent, it must be an execution whose history is of the form $H_{(u,v)}$ where $(u, v)$ form a sequence of dominoes.

Lemma 14. $\mathcal{I}$ is not causally consistent (resp., $CC, CM, CCr$) if and only if the PCP problem $P$ has a positive answer.

Proof. $(\Rightarrow)$ If $\mathcal{I}$ is not causally consistent (resp., $CC, CM, CCr$), it produces a history $h$ which is not causally consistent. By construction of $\mathcal{I}$, we must be of the form $H_{(u,v)}$, for some sequence of dominoes $(u, v)$. By Lemma 13, we know that $u = v$, and $(u, v)$ for a valid answer for $P$.

$(\Leftarrow)$ If $(u, v)$ form a valid answer to $P$, then $\mathcal{I}$ is not causally consistent, as it can produce an execution whose history is $H_{(u,v)}$, which is not causally consistent (resp., $CC, CM, CCr$) by Lemma 13.

D. Reduction to Control-State Reachability

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

Proof. $(\Rightarrow)$ Assume by contradiction that there is an execution $e \in \mathcal{I}$ which is accepted by $\mathcal{M}_{CC}$. We make a case analysis based on which branch of $\mathcal{M}_{CC}$ accepts $e$.

(First branch) If $e$ is accepted on state $q_{err}$, then it has a $rd(x) > 1$ operation with no corresponding $wr$ operation. It therefore contains bad pattern ThinAirRead, and $e$ is not $CC$.

(Second branch) Otherwise, $e$ is accepted on state $q_{err}$. Let $w_1$ be the $wr(x) > 1$ operation read after $q_1$. Let $w_2$ be the $wr(x) > 2$ operation read after the first causal link. Let $r_1$ be the $rd(x) > 1$ operation read just before $q_{err}$. Let $e_* \in \mathcal{I}$ be a differentiated execution and $f$ a renaming such that $e = e_*[f]$. Let $\mathcal{I}_* \in \mathbb{N}$ be the data value of $r_1$ in $e_*$. The renaming $f$ maps $\mathcal{I}_*$ to 1.

We show that, in the execution $e_*$, the operations $w_1$, $w_2$, and $r_1$ form bad pattern WriteCORead because $w_1 <_{co} w_2 <_{co} r_1$, $w_1 <_{rf} r_2$, and $var(w_1) = var(w_2)$. The conditions $var(w_1) = var(w_2)$, and $w_1 <_{rf} r_2$ hold, because these three operations all operate on the same variable (ensured by the register $reg_e$), and $w_1$ and $r_2$ use the same data value 1.

The condition $w_2 <_{co} r_1$ holds because either $r_1$ is between $q_2$ and $q_{err}$, in which case $reg_e$ ensures that it is on the same site as $w_2$; or $r_1$ is between $q_0$ and $q_{err}$, and in that case the precedence $rd(x) > 2$ operation makes a causality link with $w_2$.

The causality links $w_1 <_{co} w_2 <_{co} r_1$ are ensured by the presence of the CausalLink subautomata. CausalLink recognizes unbounded chains in the PO $\cup$ RF relation. In CausalLink, a $rd$ operation read from $q_0$ to $q_2$ reads-from a preceding $wr$ operation from $q_0$ to $q_0$ (thanks to register $reg_e$). Moreover, the register $reg_e$ ensures that a $rd$ operation is before, in the program order, the $wr$ operation which is following it.

We conclude by Lemma 6 that $e_*$ is not causally consistent.

(Third branch) Similar to the previous branch, but for bad pattern WriteCOInitRead.

$(\Leftarrow)$ Assume by contradiction that there is an execution $e \in \mathcal{I}$ which is not causally consistent. By Lemma 5, we can assume that $e_{*}$ is differentiated. Using Lemma 6, we have four bad patterns to consider.

(CyclicCO) The first case is when there is a cycle in PO $\cup$ RF. Without loss of generality, we can assume that the cycle is an alternation of PO and RF edges, of the form $(n > 1)$:

$$r_1 <_{PO} w_2 <_{RF} r_2 <_{PO} w_3 \ldots <_{RF} r_{n-1} <_{PO} w_n <_{RF} r_n = r_1.$$ 

This is true for two reasons. First, PO is transitive, so two PO edges can always be contracted to one. Second, RF connects wr to rd operations, so there cannot be two RF edges one after the other.

Consider the minimal prefix $e_{*}'$ of $e_{*}$ which contains only one out of these $2 \times (n - 1)$ operations. This operation must be a rd operation, as every wr operation $w_i$ is preceded by $r_{i-1} <_{PO} w_i$ in the program order. Note that $e_{*}'$ belongs to $\mathcal{I}$, as $e_{*}'$ is prefix-closed.

The execution $e_{*}'$ thus contains a rd operation $r$ which has no corresponding wr operation anywhere else in the execution, as its corresponding wr operation was among the two $2 \times (n - 1)$ operations above, and was not kept in $e_{*}'$.

Consider the renaming $f$ which maps the data value of $r$ to 1, and every other value to 5. By data independence, $e_{*}'[f]$ can be recognized by (the first branch of) $\mathcal{M}_{CC}$.

We thus obtain a contradiction, as $\mathcal{I} \cap \mathcal{M}_{CC}$ is not empty.

(ThinAirRead) The second case is when there is a rd operation with no corresponding wr operation. Again, such an execution can be recognized by the first branch of $\mathcal{M}_{CC}$ (after renaming).
(WriteCORead) The third case is when there are operations $w_1$, $w_2$, and $r_1$ in $e_s$ such that $w_1 \prec_{RF} r_1$, $\var(w_1) = \var(w_2)$, and $w_1 \prec_{CO} w_2 \prec_{CO} r_1$.

Consider the renaming $f$ which maps:
• the data value of $w_1$ and $r_1$ to 1,
• the data value of $w_2$ to 2,
• maps any value which appears in a $\var$ operation in a causality chain between $w_1 \prec_{CO} w_2$ to 3,
• maps any value which appears in a $\var$ operation in a causality chain between $w_1 \prec_{CO} w_2$ to 4,
• maps any other value to 5.

Then, $e_s[f]$ can be recognized by (the second branch of) $M_{CC}$, and $\mathcal{I} \cap M_{CC}$ is not empty.

(WriteCOInitRead) This bad pattern can be treated similarly to the previous one, but using the third branch of $M_{CC}$ instead of the second.

E. $CCv$ Bad Patterns

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.

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

($\Rightarrow$) Assume that $h$ is $CCv$, and let $co \subseteq arb$ be relations satisfying the properties of $CCv$. By Lemma 2, we know that $h$ is $CC$. Assume by contradiction that $h$ contains bad pattern CyclicCF.

Consider any edge $w_1 \prec_{CF} w_2$ in the $CF$ relation, where $\ell(w_1) = \var(x, d_1)$ and $\ell(w_2) = \var(x, d_2)$ for some $x \in X$ and $d_1 \neq d_2 \in \mathbb{N}$.

By definition of $CF$, we have $w_1 <_{CO} r_2$, where $\ell(r_2) = \text{rd}(x) > d_2$.

Moreover, $CCv$ ensures that there exists $\rho_{r_2} \in S_{RW}$ such that (CausalPast$(r_2)$, arb, $\ell$)$(r_2) \leq \rho_{r_2}$.

Since both $w_1$ and $w_2$ are in $\rho_{r_2}$, $w_1$ must be before $w_2$ in $\rho_{r_2}$, as $r_2$ is the last operation of $\rho_{r_2}$, (and $h$ is differentiated).

As a result, $w_1 <_{arb} w_2$, and $CF \subseteq arb$. The cycle in $CF \cup CO$ thus induces a cycle in $arb \cup CO$, which contradicts the fact that $CO \subseteq CO \cup arb$ and that $arb$ is strict total order.

($\Leftarrow$) Assume that $h$ is $CC$ and does not contain bad pattern CyclicCF. We use the causal order $co = CO = (PO \cup RF)^+$ to show that $h$ is $CCv$ with respect to $S_{RW}$. We must also construct the arbitration order $arb$, which is a strict total order over $O$.

We define $arb$ as any strict total order which contains $CF \cup CO$.

This is possible since $CF \cup CO$ is acyclic ($h$ does not contain bad pattern CyclicCF).

Let $r \in O$ be a read operation. We prove that there exists $\rho_r \in S_{RW}$ such that $\text{CausalArb} (r) \{r\} \leq \rho_r$.

In the case that $r$ returns the initial value 0, and because $h$ does not contain bad pattern WriteCOInitRead, there is no write on the same variable as $r$ in CausalPast$(r)$. The sequence $\rho_r$ can thus be defined as adding appropriate values to the reads different from $r$ in CausalArb$(r) \{r\}$ (that is, the value of the preceding write on the same variable, or the initial value 0 if there is no such write).

If $r$ returns a value different that 0, we know that there is a corresponding write $w$ in CausalPast$(r)$. Consider any write operation $w' \neq w$ in CausalPast$(o)$ which is on the same variable as $o$. By definition of the conflict relation $CF$ and by definition of $arb$, we know that $w' <_{arb} w$. Thus, the last write operation on variable in CausalArb$(r) \{r\}$ must be $w$. As previously, we can thus define $\rho_r$ as CausalArb$(r) \{r\}$ where we add appropriate return values to the reads different that $r$. 

$\blacksquare$
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.

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

($\Rightarrow$) Assume that $h$ is $CM$. By Lemma 1, we know that $h$ is $CC$. Assume by contradiction that $h$ contains bad pattern WriteHBInitRead or CyclicHB for some operation $o$.

By $CM$, there is $\rho_o \in S_{RW}$ with $CausalHist(o) \cup \{POPast(o)\} \leq \rho_o$. This implies in particular that the return values of all read operations which are before $o$ (in POPast($o$)) are still present in $\rho_o$.

They are not abstracted away by the projection $CausalHist(o) \{POPast(o)\}$. This can be written as $\rho_o = \rho_o'$ with $o$ added at the end. We obtain that $\rho_o \in S_{RW}$.

1) $o$ is a write operation. Here, the causal past of $o$ is the causal past of $o'$ where $o$ has been added as a maximal operation. The reason is that CO is defined as $(PO \cup RF)^*$ and the read-from relation RF only relates writes to reads. Thus, there cannot exist an operation $o''$ such that $o'' \triangleleft CO o$ and $o'' \triangleleft CO o'$. And we can define $\rho_o'$ as $\rho'$ with $o$ added at the end. We obtain that $\rho_o \in S_{RW}$.

2) $o$ is a read operation $rd(x) > 0$ for some variable $x \in X$. The fact that $h$ does not contain bad patterns WriteCOInitRead ensures that the causal past $o$ does not contain write operations on variable $x$. As in the previous case, the causal past of $o$ is the causal past of $o'$ where $o$ has been added as a maximal operation. We can thus define $\rho_o$ as $\rho'$ with $o$ added at the end. We obtain that $\rho_o \in S_{RW}$.

3) $o$ is a read operation $rd(x) > d$ for some $x \in X$ and $d \neq 0$. The fact that $h$ does not contain bad patterns ThinAirRead and WriteCORead ensures that there exists a corresponding $w$ operation such that $w \triangleleft RF o$ (in the causal past of $o$), and such there is no $w$ operation with $w < CO w_2 < CO o$ and $\text{var}(w) = \text{var}(w_2)$.

We consider two subcases:

a) $w$ is in the causal past of $o'$. By definition of $HB_o$, for any $w \in \text{CausalPast}(o)$ with $w \neq w'$ and $\text{var}(w) = \text{var}(w')$, we have $w' \triangleleft HB_o w$. The last write operation on variable $x$ in $\rho'$ must thus be $w$.

Moreover, the causal past of $o$ is the causal past of $o'$ where $o$ has been added as a maximal operation. We can thus define $\rho_o$ as $\rho'$ with $o$ added at the end. We obtain that $\rho_o \in S_{RW}$, as $o$ can read the value written by $w$.

b) $w$ is not in the causal past of $o'$. This implies that $o$ is the only $rd(x) > 0$ operation $r'$, we would have $w \triangleleft RF r' \triangleleft PO o$, and $w$ would be in the causal past of $o'$.) Let $O''$ be the set of operations contained in the causal past of $o$ (including $o$), but not contained in the causal past of $o'$. That is, $O'' = \text{CausalPast}(o) \setminus \text{CausalPast}(o')$. With the absence of bad pattern CyclicHB, we know that $w$ is a maximal (in the $HB_o$ order) write operation on variable $x$ in $o''$ (there is no write operation $w_2$ on variable such that $w < CO w_2 < HB_o w$, otherwise, by definition of $HB_o$, we would have $w_2 \triangleleft HB_o w$ and $HB_o$ would be cyclic).

We can thus define a sequence $p'' \in S_{RW}$ such that the last write operation on variable $x$ is $w$, and such that $p'' \cdot p'''$ respect the order $HB_o$. We then define $\rho_o$ as $\rho' \cdot p'''$, while setting the return values of all reads which are not in $p$ to the last corresponding write in $\rho_o$ (these can be freely modified, as they are hidden by the projection $\text{CausalHist}(o) \{POPast(o)\}$). We thus obtain that $\text{CausalHist}(o) \{POPast(o)\} \leq \rho_o$, and $\rho_o$ respects the order $HB_o$.

We consider three cases: