Dynamic Partial Order Reduction for Checking Correctness Against Weak Isolation Levels

ENRIQUE ROMÁN-CALVO, IRIF, University of Paris & CNRS, France

AHMED BOUAJJANI, IRIF, University of Paris & CNRS, France

CONSTANTIN ENEA, LIX, École Polytechinque & CNRS, France

Modern applications, such as social networking systems and e-commerce platforms are centered around using large-scale databases for storing and retrieving data. Accesses to the database are typically enclosed in transactions that allow computations on shared data to be isolated from other concurrent computations and resilient to failures. Modern databases trade off isolation for performance. The weaker the isolation level, the more behaviors a database is allowed to exhibit and it is up to the developer to ensure that their application can tolerate those behaviors.

In this work, we propose a stateless model checking algorithm for studying correctness of such applications that relies on dynamic partial order reduction. This algorithm works for a number of widely-used weak isolation levels, including Causal Consistency, Read Committed, and Read Atomic. We show that it is complete, sound and optimal, and runs with linear memory consumption in all cases. We report on an implementation of this algorithm in the context of Java Pathfinder applied to a number of challenging applications drawn from the literature of distributed systems and databases.

ACM Reference Format:

Enrique Román-Calvo, Ahmed Bouajjani, and Constantin Enea. 2022. Dynamic Partial Order Reduction for Checking Correctness Against Weak Isolation Levels. *Proc. ACM Program. Lang.* 1, 1 (June 2022), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Programming paradigm is in constant evolution, sequential programs tend to easily become obsolete because of its slow performance and even concurrent programs can also be inefficient when the memory requirements increase. The current state-of-the-art tries to overcome those problems by developing parallel programs along with distributed storage systems. However, not every type of application has the same data reliability requirements and therefore developers may want to relax the *isolation level*, i.e. the restrictions imposed to the information stored for guaranteeing consistency, from the database in order to increase performance.

Authors' addresses: Enrique Román-Calvo, IRIF, University of Paris & CNRS, France, calvo@irif.fr; Ahmed Bouajjani, IRIF, University of Paris & CNRS, France, abou@irif.fr; Constantin Enea, LIX, École Polytechinque & CNRS, France, cenea@irif.fr.

2475-1421/2022/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Allowing multiple behaviors in these contexts hinder the already difficult task of verifying concurrent programs. Studying every alternative is something unrealistic, as the number of possible scenarios grows exponentially with the length of the programs. In general, formal methods such as cite examples are a reasonable approach as they provide certificate of correctness and explainability of the bugs otherwise. Among them, *stateless model checking* (SMC) and *dynamic partial order reduction* (DPOR) cite papers stand out as the most promising techniques for verifying current programs during the recent years cite papers.

On one hand, for a given length-bounded program, SMC explores systematically every possible execution without storing at any point the set of already visited ones. On the other hand, DPOR resumes every possible behavior in a more succinct way, reducing the number of executions that have to be explored for covering those behaviors. Henceforth, combining both techniques to obtaining sound, complete and efficient algorithms has been one of the aimed goals in this field and it has been successfully done for concurrent programs with shared memory cite papers.

Despite their popularity, there is no application of such techniques in parallel programming with distributed memory's literature so far, hence the relevance of filling this gap. Nevertheless, part of the path this paper wants to create is already explored, as shared memory models are not that unrelated with distributed database's. For example, we can mention the relation between *sequential consistency* and *serializability* or *strong release-acquire* and *causal consistency*; where both database isolation level cases are nothing but a generalization of their shared memory counterparts cite papers.

In this paper, we present STMC, a *sound*, *complete*, *optimal* DPOR algorithm with *linear memory requirements* that employs SMC techniques for verifying some isolation levels. We describe the models that can guarantee those properties, show that *causal consistency* (CC), *read atomic* (RA) and *read committed* (RC) satisfy them and present an example of why more complex models such as *serializability* (SER) cannot be verified with our algorithm. We also present a formal semantics for STMC and exhibit how it evolves from the base algorithm to its current state; requiring executing transactions in isolation and swapping complete blocks of transactions. In addition, we provide some proofs to help the reader having a better understanding of STMC.

On top of this theoretical development, we also furnish this work with an implementation using Java and several benchmarks that study its re. In a nutshell, our software is an extension of JPF cite tool, a Java-built software analysis framework for Java (parallel) programs. It provides control to DFS traversal of executions, which along its modularity, makes it an ideal tool for developing and extending a database concurrent programs' verifier. In particular, we highlight the easiness for splitting the program memory and database's management and providing an API for writing the programs to analyze.

2 DEFINITIONS

In this section we describe the basic concepts STMC is built on along with the assumptions required for its correctness.

Definition 2.1. An *event* e is a tuple $\langle id, t \rangle$ where id is its *identifier* and t its *type*. The set of all events will be denoted $\mathcal{E}$.

Intuitively, an event e represents an instruction on the program and its identifier is the line number this instruction has on it. However, this description forces us to have a wide range of possible types; one per instruction types. As we want to model check transactional programs, only

those instructions related with our database are actually meaningful. Therefore, an event e would represent a succession of local instructions followed by a database instruction; instruction of type `begin`, `end`, `write` or `read`.

Further, we define a function called *variable*, $\mathtt{var} : \mathcal{E} \to \mathcal{V}$, which if e's type is `write` or `read` returns the variable it writes/reads in the database and another function called *value*, $\mathtt{val} : \mathcal{E} \to \mathbb{N}$, that for every `write` w event returns the value that $\mathtt{var}(w)$ will store after its execution. Note that for simplicity we assume $\mathtt{val}(\mathcal{E}) \subseteq \mathbb{N}$, but the actual value of the instruction could be any computable binary string; string representable in $\mathbb{N}$. Nonetheless, for clarity during our examples we may write " $a \leftarrow \mathtt{read}(x)$" or "$\mathtt{write}(x, 0)$" as a succinct notation for indicate an event's type, its variable or its value.

Definition 2.2. A *transaction* T is a finite sequence of events totally ordered by po_T where $\min_{\mathrm{po}_T} T$ has type `begin`, $\max_{\mathrm{po}_T} T$ has type `end` and every other event in T is either a `read` or `write` event. Any po-closed prefix of a transaction is called *pending transaction*.

During the whole paper we will assume that every pair of transactions are disjoint and that both $\mathcal{E}$ and $\mathcal{T}$ are finite. Therefore, we denote the function $tr : \mathcal{E} \to \mathcal{T}$ that associates every event to the unique transaction it belongs to.

Definition 2.3. A *program* $\mathcal{P}_{n,\mathcal{T}}$ is the collection of n parallel threads that execute each a sequence of transactions.

In what follows, we will assume a fixed program $\mathcal{P}$ in order to slightly relax the notation. We also assume that every transaction is included in exactly one thread, so we can map every event to the thread it belongs to via the function $\mathtt{th}_{\mathcal{P}} : \mathcal{E} \to \mathbb{Z}_n$.

For representing executions in a program we rely on the definition 2.4, which allow us representing executions as execution graphs cite paper where the vertices are transactions and the edges are their relations.

Definition 2.4. A *history* $h = \langle T, \mathrm{so}, \mathrm{wr} \rangle$ is a tuple composed by a set of (pending) transactions T (constructed over an event set $E \subseteq \mathcal{E}$) along with a strict partial order so called *session order* and a relation $\mathrm{wr} \subseteq \mathtt{writes}(E) \times \mathtt{reads}(E)$ called *write-read* s.t.

- wr^{-1} is a total function,
- so corresponds to the strict partial order defined by every thread in $\mathcal{P}$ restricted to T,
- $\mathrm{so} \cup \mathrm{wr}$ is acyclic.

We denote by $\emptyset$ the history with no vertices and we also write, with an abuse of notation, $T\ [\mathrm{wr}]\ T'$ whenever $\exists w, r \in T \times T'$ such that $w\ [\mathrm{wr}]\ r$. Finally, we briefly define several type of histories in next definition:

Definition 2.5. Let h be a history:

- h is called *complete* if every transaction is non-pending and *incomplete* otherwise;
- h is *executed in isolation* if it contains at most one pending transaction;
- h is called *total* if it is complete and contains every transaction $T \in \mathcal{T}$.

~~If the event $e = \text{NEXT}(h)$ is begin, write or end, we will denote by $h \bullet e$ the history $h' = \langle E', so', wr \rangle$ where $E' = events(h) \cup \{e\}$ and $so' = so \cup \{\langle e', e \rangle \mid e' \in h \wedge th(e) = th(e')\}$. On the other hand, if e is a read event, we will define the history $h'_w = h \bullet_w e$ for some write event $w \in h$ as $\langle E', so', wr \cup \{\langle w, r \rangle\}\rangle$; where E' and so' defined as before.~~: Not well defined, just cut-pasted from below.

EXTENSIONS, $\mathcal{H}^<$ AND $\bullet$ OPERATOR NOT (yet) DEFINED IN THIS SECTION!!!!!!!

To fully model any behavior of a transactional concurrent program we are obliged to formally describe the database section. This notion will be depicted as the concept of *model*:

Definition 2.6. An axiomatic *model* $\mathcal{M}$ over histories is a collection of rules that enforce a *consistency criterion* over them. The histories that satisfy those criteria are called $\mathcal{M}$*-consistent* while the rest are simply denoted $\mathcal{M}$-inconsistent. If there is no ambiguity on the model, we will simply denote them consistent or inconsistent.

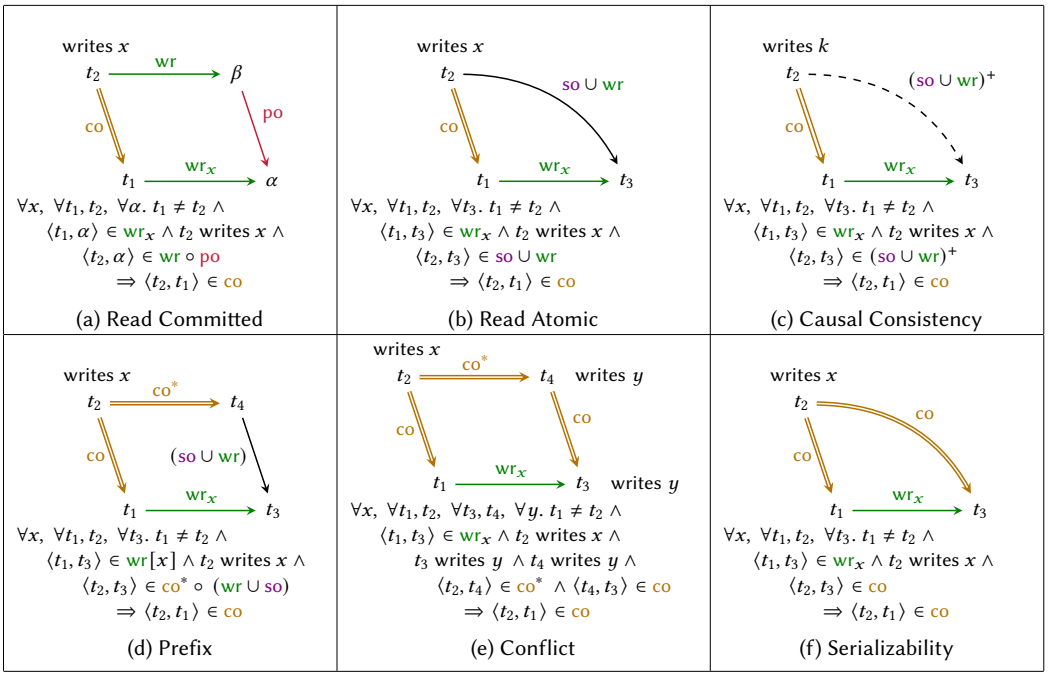
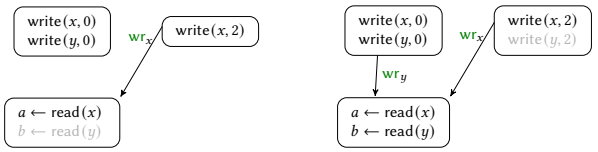


Fig. 1. Axioms defining isolations levels. The reflexive and transitive, resp., transitive, closure of a relation rel is denoted by rel^*, resp., rel^+. Also, $\circ$ denotes the composition of two relations, i.e., $rel_1 \circ rel_2 = \{\langle a, b \rangle | \exists c. \langle a, c \rangle \in rel_1 \wedge \langle c, b \rangle \in rel_2\}$.

In figure 1 it is depicted five axioms which correspond to their homonymous isolation levels: *Read Committed* (RC), *Read Atomic* (RA), *Causal Consistency* (CC) *Prefix Consistency* (PRE) and *Serializability* (SER); along with the conflict axiom. Conflict and Prefix allow us to define *Snapshot Isolation* (SI) as the model where prefix and conflict axioms both hold. We say a history h satisfies an isolation level I if there is a total order called *commit order* co that extend so ∪ wr and satisfies its axioms. However, by the definition of RC, RA and CC, it is clear that for every history h s.t. the relation co deduced from so ∪ wr is acyclic exists a commit order for those isolation levels.

3 CAUSALLY CLOSED MODELS

Besides models presented in figure 1, others isolation levels exists in literature and real life applications cite Constantin's papers + Twitter, shoppingcart.... However, our algorithm can not be analyzed under an arbitrary model. We characterize in this section the ones that can be employed by our algorithm.



(a) Extensible history. (b) Non-extensible history.

Fig. 2. Example of a dead-lock after swapping two events.

Let's analyze the histories h_1 and h_2 described in figure 2a and 2b respectively under RA; isolation level under which both are consistent. h_1 can be extended adding the event $r_1 = b \leftarrow \mathsf{read}(y)$ and the wr-edge w_1 [wr] r_1, where $w_1 = \mathsf{write}(x, 0)$. However, this is not the case of h_2: the only event that could be added in it is $w_2 = \mathsf{write}(y, 2)$. If w_2 would be added in h_2, any relation extending so $\cup$ wr and satisfying RA would be cyclic, so it wouldn't be a commit order. The essential difference between these two histories is the following: in h_1, $\mathsf{tr}(r)$ is so $\cup$ wr-maximal while in h_2 $\mathsf{tr}(w)$ is not. As real database executions forbid transactions reading from non-committed ones, it is reasonable to allow those transactions so $\cup$ wr-maximal to be executed completed without hindering the previous committed transactions.

Definition 3.1. A model $\mathcal{M}$ is called *causally closed* if for every program $\mathcal{P}$ the following conditions are satisfied:

- **Prefix-closedness:** Every so $\cup$ wr-prefix-closed sub-history of a consistent history is also consistent.
- **Causal-extensibility:** Every non-total consistent history h can be consistently extended by executing an event from a so $\cup$ wr-maximal pending transaction T in h. Moreover, if that event r is a `read`, it can always read from a `write` event w s.t. in h $\mathsf{tr}(w)$ $[\text{so} \cup \text{wr}]^*$ $\mathsf{tr}(r)$.

Comparing to the model requirements described in cite viktor's algorithm, it is causal-extensibility property, a slightly stricter version of Viktor's maximal-extensibility, the biggest difference. However, this weak formulation still forbids some axiomatic models such as Serializability cite constantin's paper (SER) Appendix cite.

Theorem 3.2. *Every model depicted in figure 1 is prefix closed.*

Proof. Let h be a consistent history. As any so $\cup$ wr-prefix-closed sub-history h' of h is a sub-graph of it, and there is a commit order co for h, it suffices to restrict co to h' for obtaining a commit order for h'. □

Theorem 3.3. *Causal Consistency (CC), Read Atomic (RA) and Read Committed (RC) are causal-extensible models.*

Proof. Let $\mathcal{M}$ a model in $\{\mathtt{CC}, \mathtt{RA}, \mathtt{RC}\}$, h a non-total consistent history and let e a so $\cup$ wr-maximal event. If e is a `begin` event, $h \bullet e$ is consistent as if there exists a commit order co for h, the relation $\mathrm{co}' = \mathrm{co} \cup \{\langle T, \mathrm{tr}(e)\rangle, T \in h\}$ is a commit order to h'. Moreover, if e is either a `write` or an `end` event, $h \bullet e$ is edge-wise identical to h, so the commit order for h is also a valid commit order for $h \bullet e$. Therefore, let e a so $\cup$ wr-maximal `read` event that reads variable x and let us find a `write` event w s.t. $\mathrm{tr}(w)$ $[\mathrm{so} \cup \mathrm{wr}]^*$ $\mathrm{tr}(r)$ and that $h'_w = h \bullet_w r$ is consistent.

For doing so, we will do an induction on the number of co cycles h'_w has, for some event w s.t. $\mathrm{tr}(w)$ $[\mathrm{so} \cup \mathrm{wr}]^*$ $\mathrm{tr}(r)$; where co Clearly, if h'_w is acyclic, by theorem cite theorem h acyclic => exists a co (another paper, I hope it exists somewhere), it is consistent. Hence, let's suppose that if h'_w has at most n cycles, there exists another `write` event w_n s.t. r causally depends on and $h'_{w_n} = h \bullet_{w_n} r$ is consistent; and let's analyze if the same property can be deduced for a history $h'_{w_{n+1}} = h \bullet_{w_{n+1}} r$ with $n+1$ cycles. As h is consistent and $\mathrm{tr}(w_{n+1})$ $[\mathrm{so} \cup \mathrm{wr}]^*$ $\mathrm{tr}(r)$, if there were a cycle, it would be due to a transaction T such that writes x, $\mathrm{tr}(w_{n+1})$ $[\mathrm{co}]^*$ T and $\varphi_{\mathcal{M}}(T, e)$, where $\varphi_{\mathtt{CC}}(T, e) = T$ $[\mathrm{so} \cup \mathrm{wr}]^+$ $\mathrm{tr}(e)$, $\varphi_{\mathtt{RA}}(T, e) = T$ $[\mathrm{so} \cup \mathrm{wr}]$ $\mathrm{tr}(e)$ and $\varphi_{\mathtt{RC}}(T, e) = T$ $[\mathrm{wr} \circ \mathrm{po}]$ e. In particular, for any of the three models, T $[\mathrm{so} \cup \mathrm{wr}]^*$ $\mathrm{tr}(r)$. Let w_n a `write` event in T that writes x and $h_{w_n} = h \bullet_w r$: if we prove that the number of co-cycles in h_{w_n} is strictly smaller than in $h_{w_{n+1}}$ by induction hypothesis, we can conclude the result.

Firstly, as T $[\mathrm{wr}_x]$ $\mathrm{tr}(r)$, the cycle that was between T and $\mathrm{tr}(w_{n+1})$ does not exist in h_{w_n}. Moreover, analogously as before, if there is a cycle in h_{w_n}, it is due to the existence of a transaction T' s.t. T $[\mathrm{co}]$ T' and $\varphi_{\mathcal{M}}(T', r)$. Therefore, in $h_{w_{n+1}}$ $\mathrm{tr}(w_{n+1})$ $[\mathrm{co}]$ T $[\mathrm{co}]$ T', so there is a cycle between in $\mathrm{tr}(w_{n+1})$ and T'. To sum up, every cycle in h_{w_n} has a counterpart in $h_{w_{n+1}}$ and it has one less cycle; so the inductive step holds.

Finally, as **`init`** contains a `write` event $w_\emptyset$ that writes x, **`init`** $[\mathrm{so} \cup \mathrm{wr}]^*$ $\mathrm{tr}(r)$ and every history has a finite number of transactions, we can deduce from the history $h'_{w_\emptyset} = h \bullet_{w_\emptyset} r$ that there is a `write` event w that r depends causally on and $h \bullet_w r$ is consistent. □

Theorem 3.4. *Prefix Consistency (*PRE*) is a causal-extensible model.*

Proof. Let h a non-total consistent history and e a so $\cup$ wr-maximal event. Analogously as in theorem 3.3, if e is a `begin` or an `end` event, $h \bullet e$ is consistent. If e is a `write` event that writes a variable x, let us show that for $h' = h \bullet e$ we can produce a commit order for it starting from a commit order co for h. If there is a cycle in h' it is because there exists some transactions T_0, T_1, T_2 s.t. T_1 $[\mathrm{wr}_x]$ T_2, T_0 $[\mathrm{so} \cup \mathrm{wr}]$ T_2, $\mathrm{tr}(e)$ $[\mathrm{co}]^*$ T_0 and T_1 $[\mathrm{co}]$ $\mathrm{tr}(e)$. As e is so $\cup$ wr-maximal, for every other transaction T, $\neg(\mathrm{tr}(e)$ $[\mathrm{so} \cup \mathrm{wr}]^*$ $T)$. Let $\mathrm{co}' = \{\langle T, T'\rangle \mid T \times T' \in h'^2$ s.t. T $[\mathrm{co}]$ $T' \wedge T \neq \mathrm{tr}(e)\} \cup \{\langle T, \mathrm{tr}(e) \mid T \in h'\rangle\}$. This relation is a total order as it is co $\restriction_{h \setminus \mathrm{tr}(e)}$ juxtaposed with $\mathrm{tr}(e)$ which extends so $\cup$ wr. Moreover, if h' had a co$'$-cycle, as $\mathrm{tr}(e)$ is so $\cup$ wr-maximal, it would have to be due to other four transactions distinct from $\mathrm{tr}(e)$; contradicting h is consistent. Therefore, h' is a PRE-consistent history.

Otherwise, if e is a `read` event that reads variable x, let $h_{R|W}$ the history obtained by splitting every transaction T in two, one containing every `read` event (R_T) that immediately so-precedes a complementary one with every single `write` event (W_T); along with corresponding $\mathrm{so}_{R|W}$ and $\mathrm{wr}_{R|W}$ relations Cite Ranadeep's paper. By theorem Cite again Ranadeep's paper, for every history $\hat{h}$, $\textsc{isConsistent}_{\mathtt{PRE}}(\hat{h})$ if and only if $\textsc{isConsistent}_{\mathtt{SER}}(\hat{h}_{R|W})$. Therefore, as $\textsc{isConsistent}_{\mathtt{SER}}(h_{R|W})$ holds, there is a commit order co$'$ such that for every T s.t. if W_T $[\mathrm{co}']$ $R_{\mathrm{tr}(e)}$ them T $[\mathrm{so} \cup \mathrm{wr}]^+ \mathrm{tr}(e)$. Therefore, let w the maximum `write` event according to co$'$ before $R_{\mathrm{tr}(e)}$ that writes x and $h'_{R|W} =$

$h_{R|W} \bullet_w e$. This history is clearly serializable taking co′ as commit order. Hence, $h' = h \bullet_w e$ is PRE-consistent and $\mathsf{tr}(w)$ $[\mathsf{so} \cup \mathsf{wr}]^* \mathsf{tr}(r)$.

□

4 THE CLASS OF SWAPPING BASED ALGORITHMS

Definition 4.1. Given an algorithm A and a model $\mathcal{M}$, we say:

- A is *$\mathcal{M}$-sound* if for every program $\mathcal{P}$, every total history h computed by A is $\mathcal{M}$-consistent,
- A is *complete* if for every program $\mathcal{P}$ it computes every possible execution,
- A is *weakly optimal* if for every program $\mathcal{P}$ it computes every execution at most once,
- A is *optimal* if for every program $\mathcal{P}$ it is weakly optimal and it does not compute blocking executions (i.e. partial histories that cannot be completed).

Definition 4.2. The algorithm's class $\mathcal{O}^n_{\mathcal{M}}$ is defined as the minimal collection containing all algorithms $\mathcal{M}$-sound, complete and optimal that employ polynomial memory and allows at most n pending transactions.

Definition 4.3. The algorithm's class $\mathcal{W}^n_{\mathcal{M}}$ is defined as the minimal collection containing all algorithms $\mathcal{M}$-sound, complete and weakly optimal that employ polynomial memory and allows at most n pending transactions.

TODO: change this to not be a class...

The above-mentioned classes are, notwithstanding, quite dense and contain lots of algorithms we are not interested in (as non-DPOR algorithms for example). In particular, our work focuses on the collection of algorithms defined by the schema 1 for some functions $\mathsf{N}, \mathsf{E}, \mathsf{V}, \mathsf{C}, \mathsf{P}, \mathsf{S}$ to be defined.

Algorithm 1 EXPLORE algorithm

Input: h: history

1: $e \leftarrow \mathtt{N}(h)$
2: **if** TYPE$(e) = \bot$ **then**
3: **if** $\mathtt{E}(h)$ **then**
4: **output** h
5: **end if**
6: **return**
7: **else if** TYPE$(e) = \mathtt{read}$ **then**
8: **for all** $w \in \mathtt{V}(h, e)$ **do**
9: EXPLORE$(h \bullet_w e)$
10: **else**
11: EXPLORE$(h \bullet e)$
12: **end if**
13: $l \leftarrow \mathtt{C}(h)$
14: **for all** $(\alpha, \beta) \in l$ **do**
15: **if** $\mathtt{P}(h \bullet e, \alpha, \beta)$ **then**
16: EXPLORE$(\mathtt{S}(h \bullet e, \alpha, \beta))$
17: **end if**

Algorithm 1 explores systematically the space of histories, selecting a new event e to be added to some history h if possible, thanks to the function N called *next*. If it wasn't possible, it is due to h being either a total execution or an undesirable one; both behaviors discriminated via *evaluating* h with the E function. Otherwise, e will added h along with an eventual wr-edge obtained via V, linking a *valid* write event with the new read event. Moreover, at some point during the search traversal determined by C the algorithm will *compute* some collection of events α, β that may needed to be reordered. Every events' rescheduling possibly lead to a different execution, so for controlling which reorderings we are shall explore, we enforce a *reordering protocol* (function P). In the affirmative case, the new histories would be generated via S, *swapping* β and all their dependencies before α. However, the reader shall take into account that this high-level description of algorithm 1 may not be satisfied for some EXPLORE's instance.

Definition 4.4. The *swapping-based algorithm's class* for the memory model $\mathcal{M}$, $\mathcal{S}_{\mathcal{M}}$, is the minimal collection containing all algorithms A that can be described as algorithm 1's instances: EXPLORE(N, E, V, C, P, S); where N, E, V, C, P, S are functions defined as follows:

- $\mathtt{N} : \mathcal{H}_{\mathcal{P}} \to \mathcal{E}_{\mathcal{P}}$, where for every $h \in \mathcal{H}_{\mathcal{P}}$, $\mathtt{N}(h) \notin h$ and for every event e s.t. e [so] $\mathtt{N}(h)$, $e \in h$,
- $\mathtt{E} : \mathcal{H}_{\mathcal{P}} \to \{0, 1\}$,
- $\mathtt{V} : \mathcal{H}_{\mathcal{P}} \times \mathcal{E}_{\mathcal{P}} \to \mathcal{P}(\mathcal{E}_{\mathcal{P}})$,
- $\mathtt{C} : \mathcal{H}^{<}_{\mathcal{P}} \to \mathcal{E}^{*}_{\mathcal{P}} \times \mathcal{E}^{*}_{\mathcal{P}}$, where for every history h, $\mathtt{C}(h) = (\alpha, \beta)$, $\alpha \cap \beta = \emptyset$ and for every $(e, e') \in \alpha \times \beta$, $e <_h e'$.
- $\mathtt{P} : \mathcal{H}^{<}_{\mathcal{P}} \times \mathcal{E}^{*}_{\mathcal{P}} \times \mathcal{E}^{*}_{\mathcal{P}} \to \{0, 1\}$,
- $\mathtt{S} : \mathcal{H}^{<}_{\mathcal{P}} \times \mathcal{E}^{*}_{\mathcal{P}} \times \mathcal{E}^{*}_{\mathcal{P}} \to \mathcal{H}^{<}_{\mathcal{P}}$, where for every h, α, β, we have $\mathtt{S}(h \bullet e, \alpha, \beta) = h'$, $\alpha, \beta \in h'$, for every $(e, e') \in \alpha \times \beta \Rightarrow e >_{h'} e'$ and there exists some $(e, e') \in (\alpha, \beta)$ s.t. $\mathrm{tr}(e')$ [wr] $\mathrm{tr}(e)$.

The swapping-based algorithms have been already studied in the literature as for example cite Viktor's algorithm, which belongs to $\mathcal{S}_{SC}$; where SC in this case is the axiomatic representation of sequential consistency memory model. cite SC.

5 A STATELESS DPOR ALGORITHM FOR CAUSALLY-CLOSED MODELS

The main goal in this section is describing a deterministic algorithm for transactional model-checking under a causally-closed model $\mathcal{M}$ that obtains all possible behaviors a program may have. We present during this section a swapping-based sound, complete and optimal algorithm employing polynomial memory. In particular, we will show that our algorithm has at most one pending transaction and why this is key to guarantee the rest of the properties. For the legibility of this document, we will postpone their proofs to section 7.

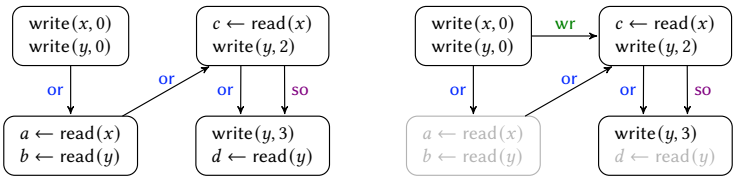
5.1 Oracle and algorithm order

The version here presented employs as parameter the program to analyze along with a total order called *oracle order* between its transactions. This order, denoted as $\leq_{\text{or}}$ and trivially extensible to the events, has to respect the session order of the program (i.e. if T [so] T' then $T \leq_{\text{or}} T'$); forbidding executions any real processor would produce. This order will be constant during the whole algorithm's execution.

In addition, we assume the algorithm maintains a total order between the events in every history, called *algorithm order* and denoted as $\leq_h$, as well as a function NEXT that given a non-total history h returns the next event to be added. In a nutshell, it returns the minimal event according to or that is not in h, prioritizing those events in pending transactions. Formally:

$$\textsc{next}(h) \quad := \quad \begin{cases} \min_{\text{or}}\{e \in \mathcal{E} \mid e \notin h\} \cup \{\bot\} & \text{if } \nexists T \text{ s.t. } \textsc{pending}_h(T) \\ \min_{\text{or}}\{e \in \mathcal{E} \setminus h \mid e \in \textsc{pending}_h(T)\} & \text{otherwise} \end{cases}$$

Thanks to this function, we will be able to extend any history $h = \langle E, \text{so}, \text{wr}\rangle$ in a deterministic way. Moreover, by its definition, we observe that NEXT always propose to complete pending transactions before starting new ones. Therefore, it is a reasonable candidate as N function in a algorithm 1 instance.



(a) or produce a total order between all transactions in the program.

(b) An incomplete history and its or order.

Fig. 3. Some possible oracle order between transactions.

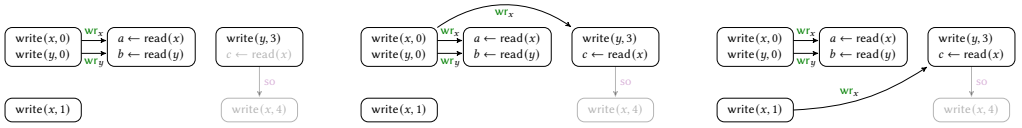
For example, given the history h in Figure 3b, the function NEXT would return the event $d \leftarrow \text{read}(y)$ instead of $a \leftarrow \text{read}(x)$; as the forth transaction is pending in h. Finally, we declare the function EVALUATE that detects when a history is total.

$$\textsc{evaluate}(h) \quad := \quad \textsc{next}(h) = \bot$$

5.2 Extending and swapping histories

The incremental process of obtaining histories with more information it is called *extension*. In essence, given a history h and an event $e \notin h$, all possible graphs using these two elements must be constructed.

If e's type is `begin` or `end`, there is only one possible way to extend it by the operator $\bullet$'s definition. When e is a `read`, however, we have to explore multiple histories, one per wr-dependency that can be generated with a `write` event $w \in h$ and e. That is, all histories $h \bullet_w e$. For example in figure 4 we can see how from the history in figure 4a we can obtain two different histories depending on the wr dependency created (figures 4b and 4c). In both three cases, we declare the algorithm order for this new history h' by simply juxtaposing e to all the previous ones: $\leq_{h'} = \leq_h \cup \{\langle e', e\rangle \mid e' \in h\}$.



(a) Current history. (b) One possible extension. (c) Another possible extension.

Fig. 4. Extensions of a history by adding a `read` event.

However, we will only select those `write` event such that the extended history $h \bullet_w e$ is consistent, as described by function $\textsc{valid}_{\mathcal{M}}$; which will eventually play the `V`'s role in our algorithm.

$$\textsc{valid}_{\mathcal{M}}(h, e) \quad := \quad \{w \mid \textsc{isConsistent}_{\mathcal{M}}(h \bullet_w e)\}$$

Conversely, adding a `write` w is more complicated, multiple events may read this new event; specifically. the number of possible histories is exponential, $2^{|\mathtt{read}(h)|}$. Moreover, after changing a wr-edge from a `read` r, we have no knowledge of the rest of the event's presence: some conditional instruction may be executed after r and it may be meaningless talking about them. Thus, checking for every possible set of `read` events if reading from w leads to something consistent is no reasonable. Therefore, we have to define a criterion to determine whose sets of read events shall be analyzed. One hand, the history $h \bullet w$ where no `read` reads from w has to be explored, with an algorithm order defined in an analogously as for any other aforementioned history. On the other hand, we select one `read` r that will be the first event in h reading from w while the ones that follow r will have to be re-executed. As r had already been executed, what at the end we produce is a swap between the relative orders of r and w; so r would be thereinafter called *swapped*.

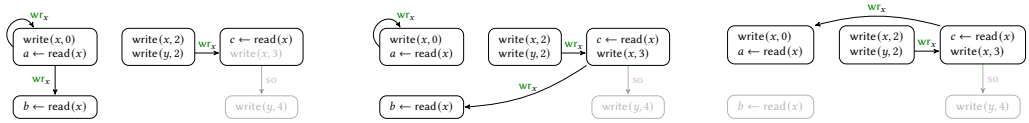
Definition 5.1. A `read` event r is *swapped* if the following conditions hold:

- For $w = h.\mathrm{wr}(r)$, $w <_h r$ and $w >_{\mathrm{or}} r$.
- $\mathrm{tr}(r)$ is the first transaction that depends on $\mathrm{tr}(w)$: $\nexists T <_{\mathrm{or}} \mathrm{tr}(r)$ s $T <_h \mathrm{tr}(r)$ and $\mathrm{tr}(w)$ $[\mathrm{so} \cup \mathrm{wr}]^+ T$.

- r is the first read event that reads from $\mathrm{tr}(w)$: $\nexists r' \in \mathrm{tr}(r), r' \leq_{or} r$ such that $\mathrm{tr}(h.\mathrm{wr}(r')) = \mathrm{tr}(w)$

This definition is summarized in the following function:

$$\textsc{swapped}(h, r) \quad := \quad \begin{array}{c} \mathtt{type}(r) = \mathtt{read} \wedge w <_h r \wedge w >_{or} r \\ \wedge \\ \forall e \in h, (\mathrm{tr}(e) <_{or} \mathrm{tr}(r)) \Rightarrow (r <_h e \vee (\neg(\mathrm{tr}(w)\ [\mathrm{so} \cup \mathrm{wr}]^+\ \mathrm{tr}(e))) \\ \wedge \\ \forall e \in \mathtt{read}(h), (e\ [\mathrm{po}]\ r) \Rightarrow \mathrm{tr}(h.\mathrm{wr}(r)) \neq \mathrm{tr}(w) \end{array}$$



(a) Current history. (b) One possible swap. (c) Another possible swap.

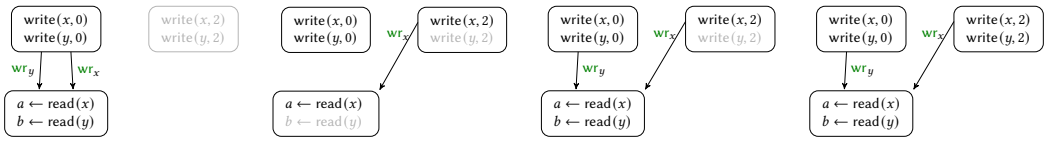
Fig. 5. Extensions of a history by adding a write event.

Let us show with an example the power of swapping. Here, in Figure 5a we can see an incomplete history whose NEXT event, $w := \mathtt{write}(x, 3)$, is a write event. There are four possible histories, depending if the first and second read events, $r_a := a \leftarrow \mathtt{read}(x)$ and $r_b := b \leftarrow \mathtt{read}(x)$, read or not from w (as $c \leftarrow \mathtt{read}(x)$ will never be able to read from w). In Figure 5b we can see the history h_b where only r_b reads from w. As $r_a <_h r_b$, we simply state that $w\ [\mathrm{wr}]\ r_b$ in h_b. The other two cases that modify the write-read relation are due to the edge $w\ [\mathrm{wr}]\ r_a$; their common root. Therefore, we can construct the history h_c depicted in Figure 5c, mark r_b as no executed to later on re-execute it and decide, in a later moment, if $w\ [\mathrm{wr}]\ r_b$ or not. In even in this small case we can realize that if h_c would be inconsistent, we would already reduce the number of explored histories by one; its extensions would also be inconsistent.

In a more general context, when swapping two events r and w, we will delete all those events e that are between r and w in the history-order such that $\mathrm{tr}(w)$ does not depends on. Otherwise, deleting some event e such that $\mathrm{tr}(e)\ [\mathrm{so} \cup \mathrm{wr}]^*\ \mathrm{tr}(w)$ holds will produce a history where either some read is reading from a deleted write or some event would be executed before its so-predecessor; both impossible situations in real life.

5.3 Avoiding inconsistent branches

Besides sound and complete, we would also seek for an optimal algorithm, i.e. that avoids computing a history h whose extensions are all inconsistent; also called a *blocking* execution. If this is not achieved, our search would employ more resources such as time or memory than it actually needs for doing its purpose.



(a) Current. (b) After swapping. (c) Extended. (d) Inconsistent.

Fig. 6. Example of a dead-lock after swapping two events.

One example of this undesired behavior can be easily seen under RA memory model with the program depicted in Figure 6. Here we consider three transactions ordered from left to right, top to bottom, and we start with the history depicted in Figure 6a. After executing the NEXT event, the `write` $w_x := \mathtt{write}(x, 2)$, one possible action would be swapping w_x with the event $a \leftarrow \mathtt{read}(x)$; obtaining the history h_2 portrayed in Figure 6b. By definition of the NEXT function, that history shall be extended with $r_b := b \leftarrow \mathtt{read}(y)$, but as there is only one `write` instruction that writes y, r_b must read from the very first transaction; as seen in Figure 6c. However, when completing the third transaction we must inexorably admit that our history in Figure 6d is inconsistent. Therefore, all the computation required to complete the unfinished transactions after the swap was in vain; we couldn't detect after computing h_2 the dead end.

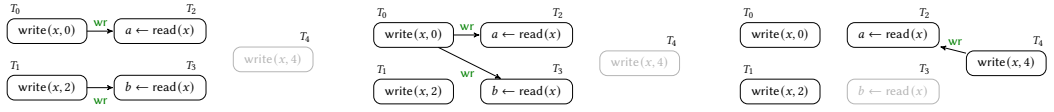
This example fails as history from figure 6c has a pending transaction that is not so ∪ wr-maximal. Hence, for being always able to extend a history by invoking the model's causally-extensibility, we have to never produce pending transactions that are non so ∪ wr-maximal. A simple solution is always executing histories in isolation, i.e. having exactly one pending transaction; thus, so ∪ wr-maximal, otherwise there would be a previous point where two pending transactions coexisted. To sum up, we are not going to swap just after executing a `write` event but when its transaction is completed.

Following algorithm 1's schema, we define two functions COMPUTE, SWAP that plays the role of `C`, `S` respectively. In addition, for the history $h' = \textsc{swap}(h, r, w)$ we declare the algorithm order of h', $\leq_{h'}$ as $\leq_{h'} = \leq_{(h \setminus \mathtt{tr}(r))\restriction_{h'}} \cup \{\langle e, e' \rangle \mid e \in h' \setminus \mathtt{tr}(r), e' \in \mathtt{tr}(r)\} \cup \mathrm{po}_{\mathtt{tr}(r)\restriction_{h'}}$.

$$
\begin{array}{lll}
\textsc{compute}(h) & := & \{(r, w) \in h^2 \mid r <_h w \wedge \mathtt{var}(r) = \mathtt{var}(w) \wedge w \in \mathtt{tr}(\mathtt{last}(h))\} \\
\textsc{swap}(h, r, w) & := & (h \setminus D) \bullet_w r \\
& \text{where} & D = \{e \mid r \leq_h e \wedge \neg(\mathtt{tr}(e)\ [\mathrm{so} \cup \mathrm{wr}]^*\ \mathtt{tr}(w))\}
\end{array}
$$

5.4 Maximally added events

In a context where no blocking branches can be produced, optimality and weakly optimality notions coincide. Moreover, every extension produces a history with more non-redundant information, so it is no a source of redundancy. However, swapping every pair of events generated by COMPUTE's function may produce duplicate explored histories.



(a) History a. (b) History b. (c) After swap history.

Fig. 7. Two different histories after swapping lead to a common one.

Let's suppose we have a program $\mathcal{P}$ as depicted in figure 7, assuming $\leq_{\text{or}}$ order transactions as presented from left to right, top to bottom. Then, the algorithm computes histories 7a, h, and 7b, h'. After adding T_4 in both h, h' we can produce a swap between $r_a := a \leftarrow \text{read}(x)$ and $w_4 := \text{write}(x, 4)$; deleting the event $r_b := b \leftarrow \text{read}(x)$ as $\neg(\text{tr}(r_b)\ [\text{wr} \cup \text{so}]^+\ \text{tr}(w_4))$. Therefore, after the swap in both cases we arrive to the history depicted in Figure 7c; obtaining a non-optimal situation.

In conclusion, we cannot swap transactions without any restriction. As the example in figure 7 shows, the key of redundancy lies in every wr edge that is going to be modified or erased: if two histories only differ on those, the resultant history is the same.

Definition 5.2. An event e is *maximally added* in a history h if $\textsc{isMaximallyAdded}_{\mathcal{M}}(h, e)$ formula is satisfied.

$$\textsc{isMaximallyAdded}_{\mathcal{M}}(h, e) \quad := \quad \texttt{type}(e) \neq \texttt{read} \vee \text{IMA}^{\text{READ}}_{\mathcal{M}}(h, e, h.\text{wr}(e))$$

$$\text{where IMA}^{\text{READ}}_{\mathcal{M}}(h, r, w) \quad := \quad \left(\begin{array}{c} \neg\textsc{swapped}(h, r) \wedge \textsc{isConsistent}_{\mathcal{M}}(h) \\ \wedge \\ \forall e \in h,\ r\ [\text{po}]\ e \Rightarrow \neg\textsc{swapped}(h, e) \\ \wedge \\ \forall w' \in h,\ w' <_h r \Rightarrow \\ (w' \leq_h w \vee \neg\textsc{isConsistent}_{\mathcal{M}}((h \setminus D) \bullet_{w'} r))) \\ \\ \text{where } D = \{e \mid r \leq_h e\} \end{array} \right)$$

Intuitively, definition 5.2 allow us to detect when a `read` event r reads from some *default* value, the last `write` w event writing x that was added before r and such that the resultant history is consistent. In general, the source of non-optimality comes from the existence of histories differing in some wr-edge involving a transaction that will be deleted. Therefore, we can stablish a simple criterion for guaranteeing optimality: a swap between two events can only happen when every event that have to be re-executed is maximally added. This criterion is defined as function $\textsc{isSwappable}_{\mathcal{M}}$ and it will play the role of P function in our algorithm 1's instance.

$$\textsc{isSwappable}_{\mathcal{M}}(h, r, w) \quad := \quad \begin{array}{c} r \in Del \wedge \forall e \in Del : \textsc{isMaximallyAdded}_{\mathcal{M}}(h, e) \\ \text{where} \quad Del \quad = \quad \{e \mid r \leq_h e \wedge \neg(\text{tr}(e)\ [\text{wr} \cup \text{so}]^*\ \text{tr}(w))\} \end{array}$$

Altogether, our swapping-based algorithm can simply being defined as an instance of 1: EXPLORE$_{\mathcal{M}}$(NEXT, EVALUATE, VALID$_{\mathcal{M}}$, COMPUTE, ISSWAPPABLE$_{\mathcal{M}}$, SWAP). A full detailed pseudocode of this procedure can be seen as algorithm 2; with initial call $h = \emptyset$.

Algorithm 2 Optimal recursive STMC

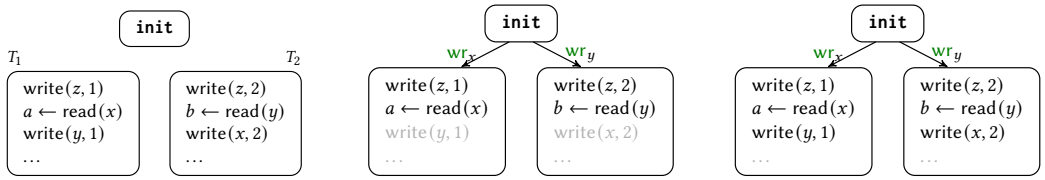
Input: h: history

```
1:  e ← (h)
2:  if e = ⊥ then
3:      if EVALUATE(h) then
4:          output h
5:      end if
6:      return
7:  else if TYPE(e) = read then
8:      for all w ∈ VALID_M(h, e) do
9:          EXPLORE_M(h ∘_w e)
10: else
11:     EXPLORE_M(h ∘ e)
12: end if

13: for all (α, β) ∈ COMPUTE(h) do
14:     if ISSWAPPABLE_M(h ∘ e, α, β) then
15:         EXPLORE_M(SWAP(h ∘ e, α, β))
16:     end if
```

6 WEAK DPOR ALGORITHMS FOR SNAPSHOT ISOLATION AND SERIALIZABILITY

As show in section 7, algorithm 2's completeness proof (theorem 7.15) is model-depending, as it heavily relies on its causal-extensibility. Immediately, the question of the algorithm's extensibility to stricter isolation levels arise. For understanding the difference between the formers and SI or SER, let's analyze how algorithm 2 behaves for the program depicted in figure 8a under them. We study this example as it has exactly two consistent executions but a dead-lock one under SI and SER.



(a) Original program. (b) Consistent partial history. (c) Dead-lock history.

Fig. 8. A program and some partial histories. In gray, instructions not yet executed.

As $\mathcal{P}$ is symmetric and every computed partial execution have at most one pending transaction, we can assume, without loss of generality, the oracle order where T_1 precedes T_2. In this setting, when A execute the read event $r_1 := a \leftarrow \text{read}(x)$, it will only be able to read from **init**. After

completing T_1, it executes the read-event $r_2 := b \leftarrow \mathtt{read}(x)$ from T_2. One one hand, if r_2 reads from T_1, then we obtain a consistent history that can be extended to a total consistent one, h. However, from h we cannot swap any two transactions as T_2 depends on T_1, which also depends on **init**. On the other hand, if r_2 reads from **init** we will obtain a history h' that leads to a deadlock (figure 8c) and cannot be swapped before T_2's completion; otherwise we would obtain a partial history h'' with two pending transaction. To sum up, algorithm 2 is incomplete for SI and SER.

This show that our proposal is not flexible enough to satisfy the constraints that new models may require. Therefore, we want to analyze what properties shall we weak for still obtaining a $\mathcal{M}$-sound, complete and optimal while employing polynomial memory; for SI and SER models. Let's suppose then that there exists such an algorithm $A \in \mathcal{S}_{\mathcal{M}}$ and see how it behaves under the same program $\mathcal{P}$. As A is complete, whenever a read event r reading from some write w does not lead to an inconsistency, it will explore that possible history (the algorithm cannot deduce in advance if a dead-lock/inconsistency will appear later on). In particular, a consistent extension of the history shown in 8b will be computed with either $\mathtt{write}(y, 1)$ or $\mathtt{write}(x, 2)$ in it. Without loss of generality, let's assume $w_2 := \mathtt{write}(x, 2) \notin h$. By definition of N function, $a = \mathtt{N}(h) \in h$, and as stated in lines 9, 11, the history $h \bullet a$ is an extension of h that will always be explored. Moreover, as A is $\mathcal{M}$-sound, $\mathtt{N}(h')$ cannot return w_2 as $\neg$ IsConsistent$_{\mathcal{M}}(h' \bullet w_2)$ for any extension h' of h. Therefore, there is a history h' that extends h where all events but those in T_2 po-after w_2 will be executed. In conclusion, $\mathtt{N}(h') = \bot$ but h' is not total, so it must be a blocking execution. That contradicts A is an optimal algorithm.

Theorem 6.1. *There is no algorithm in $\mathcal{S}_{\mathcal{M}} \cap O^n_{\mathcal{M}}$, n in $\mathbb{N}$, for neither SI nor SER.*

In particular, the program in figure 8 shows that both SI and SER are actually not causally-extensible. Nevertheless, we present a swapping-based $\mathcal{M}$-sound, complete, weakly optimal algorithm employing polynomial memory based on 2:

Present code

Theorem 6.2. *Algorithm cite SER's algo belongs to $\mathcal{S}_{\mathcal{M}} \cap \mathcal{W}_{\mathcal{M}}$ for model $\mathcal{M}$, $\mathcal{M} \in \{\mathtt{SI}, \mathtt{SER}\}$.*

Discuss the exponential total branches ditched

7 PROOF OF THE ALGORITHMS

Introduction

7.1 Soundness

Theorem 7.1. *Algorithm 2 is sound.*

Proof. We prove this theorem by induction on the number of steps a reachable history needs in a computable path to be reached. If this number is zero, the history is $\emptyset$, which is consistent; so let us prove the inductive step, assuming that in any computable path of length at most n every history is consistent. Let h computed in a $n+1$ path, and let h_p the immediate predecessor of h, which is consistent, and $a = \textsc{next}(h_p)$. If a is not a read event and $h = h_p \bullet a$, by next's definition along with the model's causal-extensibility, a is a so $\cup$ wr-maximal event so h is consistent. Otherwise, if a is a read event and $h = h_p \bullet_w a$ for some write event w, by $\textsc{valid}_{\mathcal{M}}$'s definition we know that h is consistent. Finally, if $h = \textsc{swap}(h_p, r, w)$ for some events r, w, as $\textsc{isSwappable}_{\mathcal{M}}(h_p, r, w)$ is satisfied, h is consistent. □

7.2 Completeness

In our algorithm's context, completeness means being able to compute every total history. However, our algorithm works with an extended version of histories where its events are totally ordered. For proving this property, we will need to furnish every history with a total order that coincides with the algorithm's one. This order is given by the canonical order function presented below.

Algorithm 3 CANONICAL ORDER

```
1:  procedure canonicalOrder(h, T, T')
2:      return T [so ∪ wr]* T' ∨
3:          (¬(tr(T') [so ∪ wr]* T) ∧ minimalDependency(h, T, T', ⊥)
4:  end procedure

5:  procedure minimalDependency(h, T, T', e)
6:      let a = min_{<or} dep(h, T, e); a' = min_{<or} dep(h, T', e)
7:      if a ≠ a' then
8:          return a <or a'
9:      else
10:         return minimalDependency(h, T, T', a)
11:     end if
12: end procedure

13: procedure dep(h, T, e)
14:     return {r | ∃w s.t. T [so ∪ wr]* tr(w) ∧ w [wr] r ∧ tr(r) [so ∪ wr]+ tr(e)} ∪ T
15: end procedure
```

The function canonicalOrder produces a relation between transactions in a history, denoted $\leq^h$. In algorithm 3's description, we denote $\bot$ as the end of the program, which always exists, and that is so-related with every single transaction.

Lemma 7.2. *For every history h, event e and transaction T, $\text{dep}(h, T, \min_{<_{or}} \text{dep}(h, T, e)) \subseteq \text{dep}(h, T, e)$. Moreover, if $\text{dep}(h, T, e) \neq T$, the inclusion is strict.*

Proof. Let $r' = \min_{<_{or}} \text{dep}(h, T, e)$ and $r \in \text{dep}(h, T, r')$. Then, $\exists w$ s.t. $T\ [\text{so} \cup \text{wr}]^*\ \text{tr}(w) \wedge w\ [\text{wr}]\ r \wedge \text{tr}(r)\ [\text{so} \cup \text{wr}]^+ \text{tr}(r')$ and $\exists w'$ s.t. $T\ [\text{so} \cup \text{wr}]^*\ \text{tr}(w') \wedge w'\ [\text{wr}]\ r' \wedge \text{tr}(r')\ [\text{so} \cup \text{wr}]^+ \text{tr}(e)$; so $\text{tr}(r)\ [\text{so} \cup \text{wr}]^+ \text{tr}(r')\ [\text{so} \cup \text{wr}]^+ \text{tr}(e)$. In other words, $r \in \text{dep}(h, T, e)$. The moreover comes trivially as $r' \notin \text{dep}(h, T, r')$. □

Lemma 7.3. *For every distinct T, T', minimalDependency(h, T, T', e) always halts.*

Proof. As h is a finite history, every transaction T belongs to $\text{dep}(h, T, e)$, regardless of the event e and via lemma 7.2 the set dep shrinks in each recursive call; we conclude that if $T \neq T'$, there would be a call of minimalDependency and an event e associated with it s.t. $\min_{<_{or}} \text{dep}(h, T, e) \neq \min_{<_{or}} \text{dep}(h, T', e)$. □

Lemma 7.4. *The relation $\leq^h$ is a total order.*

Proof.

- Reflexivity: By definition, for every T, $T \leq^h T$.
- Transitivity: Let's suppose $a \leq^h b$ and $b \leq^h c$. First, take into account that if $c \neq a$, $\neg(c\ [\text{so} \cup \text{wr}]^*\ a)$ Here we distinguish four cases:
  - If $a\ [\text{so} \cup \text{wr}]^*\ b$ and $b\ [\text{so} \cup \text{wr}]^*\ c$, then $a\ [\text{so} \cup \text{wr}]^*\ c$, so $a \leq^h c$.
  - If $a\ [\text{so} \cup \text{wr}]^*\ b$ but $\neg(b\ [\text{so} \cup \text{wr}]^*\ c)$, then for every $e \in h$, $\min_{<_{\text{or}}} \text{DEP}(a, e) \leq_{\text{or}} \min_{<_{\text{or}}} \text{DEP}(b, e)$, so $a <^h c$.
  - If $\neg(a\ [\text{so} \cup \text{wr}]^*\ b)$ but $b\ [\text{so} \cup \text{wr}]^*\ c$, then for every $e \in h$, $\min_{<_{\text{or}}} \text{DEP}(b, e) \leq_{\text{or}} \min_{<_{\text{or}}} \text{DEP}(c, e)$, so $a <^h c$.
  - If $\neg(a\ [\text{so} \cup \text{wr}]^*\ b)$ and $\neg(b\ [\text{so} \cup \text{wr}]^*\ c)$, then it can be proven by induction that $a <^h c$. It has to be proven iterating on the call function mininalDependency, a bit boring
- Antisymmetric For every a, b s.t. $a \leq^h b$ and $b \leq^h a$. If $a\ [\text{so} \cup \text{wr}]^* b$, then $a = b$. If not, then $\text{MINIMALDEPENDENCY}(h, a, b, \bot)$ and $\text{MINIMALDEPENDENCY}(h, b, a, \bot)$ cannot be satisfied at the same time. Again an induction on MINIMALDEPENDENCY along with the history's finiteness.
- Strongly connection Let a, b s.t. $a \not\leq_{\text{or}} b$. If $b\ [\text{so} \cup \text{wr}]^* a$, then $b \leq_{\text{or}} a$. Otherwise, as $\neg(a\ [\text{so} \cup \text{wr}]^*\ b)$ and MINIMALDEPENDENCY halts (lemma 7.3) and $\neg\text{MINIMALDEPENDENCY}(h, a, b, e)$, then $\text{MINIMALDEPENDENCY}(h, b, a, e)$; so $b <^h a$.

□

Definition 7.5. A reachable history h is or-*respectful* if it has at most one pending transaction and for every pair of events $e \in \mathcal{P}, e' \in h$ s.t. $e \leq_{\text{or}} e'$, either $e \leq_h e'$ or $\exists e'' \in h, \text{tr}(e'') \leq_{\text{or}} \text{tr}(e)$ s.t. $\text{tr}(e')\ [\text{so} \cup \text{wr}]^*\ \text{tr}(e'')$, $e'' \leq_h e$ and $\text{SWAPPED}(h, e'')$; where if $e \notin h$ we state $e' \leq_h e$ always hold but $e \leq_h e'$ never does. We will denote it by $R^{\text{or}}(h)$.

Lemma 7.6. *Every reachable history is* or*-respectful.*

Proof. We will prove it by induction on the number of 2's stack calls a computable path that leads to a history h needs, n. The base case, $n = 0$, is for the trivial history $h = \emptyset$ where it trivially holds; so let us prove the inductive case; being $e \leftarrow \text{NEXT}(h)$. On one hand, e is not a `read` nor a `begin` event and $h' = h \bullet e$, as $\neg\text{SWAPPED}(h, e)$ and h' is edge-wise identical to h, $R^{\text{or}}(h')$ holds.

If e is a `begin` event, $h' = h \bullet e$. Let $a \in \mathcal{P}, b \in h'$ s.t. $a \leq_{\text{or}} b$. If $a \in h$ or $b \neq e$, as $\leq_{h'}$ is an extension of $\leq_h$ and $R^{\text{or}}(h)$, the property holds. Moreover, as $e = \min_{\text{or}} \mathcal{P} \setminus h$, there is no event $a \in \mathcal{P} \setminus h$ s.t. $a \leq_{\text{or}} e$; so the property holds.

On the other hand, if e is a `read` event and w is a `write` one, let us prove that $h' = h \bullet_w e$. Let $a \in \mathcal{P}, b \in h'$ s.t. $a \leq_{\text{or}} b$. Once again, if $a \in h$ or $b \neq e$ the property holds; so let's suppose $a \in \mathcal{P} \setminus h$ and $b = e$. Let $d = \text{begin}(\text{tr}(e))$, $d \in h$. As $R^{\text{or}}(h)$ and $a \notin h$, $a \leq_{\text{or}} d$; so there exists $c \in h$, $\text{tr}(c) \leq_{\text{or}} \text{tr}(a)$ s.t. $\text{tr}(d)\ [\text{so} \cup \text{wr}]^*\ \text{tr}(c)$, $c \leq_h d$ and $\text{SWAPPED}(h, c)$. As $\text{tr}(r) = \text{tr}(d)$, we conclude $R^{\text{or}}(h)$.

Finally, let $h' = \text{SWAP}(h \bullet e, r, w)$ for some $r, w \in h$ s.t. $\text{ISSWAPPABLE}_{\mathcal{M}}(h \bullet e, r, w)$ holds. Let a, b two event s.t. $a \leq_{\text{or}} b$. If $a \leq_{h'} b$ or, as $R^{\text{or}}(h)$ and $\text{ISSWAPPABLE}_{\mathcal{M}}(h \bullet e, r, w)$ holds, $a \not\leq_h b$, then

the property is satisfied; so let's suppose $b <_{h'} a$ and $a \leq_h b$. In this situation, a has to be a deleted event, so $a \in \mathcal{P} \setminus h' \cup \{r\}$. As $r \leq_h a$, if $a \leq_{\text{or}} r$, there would exist a $c \in h$, $\text{tr}(c) \leq_{\text{or}} \text{tr}(a) \leq_{\text{or}} \text{tr}(r)$ s.t. $\text{tr}(r)$ $[\text{so} \cup \text{wr}]^* \text{tr}(c)$ and $\text{SWAPPED}(h, c)$. However, this contradicts $\text{isSwappable}_{\mathcal{M}}(h \bullet e, r, w)$; so $r \leq_{\text{or}} a$. Taking $e'' = r$ the property is witnessed. □

Proposition 7.7. *For any reachable history h, $\leq^h \equiv \leq_h$.*

Proof. We will prove this lemma by induction on the number of steps a computable path leading to h are required by algorithm 2. The base case, $n = 0$, implies $h = \emptyset$, so both relations hold. Let's suppose that for every history h' that requires at most n steps, $\leq^{h'} \equiv \leq_{h'}$; and let's analyze $\leq^h$ for a history computed with $n + 1$. In particular, there exists a history h_p in that path which is an immediate predecessor of h. We will distinguish cases depending on how from h_p we reach h; calling $e = \text{NEXT}(h)$

- Adding a `end`, `write`: As h_p and h are edge-wise identical, $\leq^h \equiv \leq_h$.
- Adding a `begin`: As $\text{DEP}(h_p, T, \bot) = \text{DEP}(h, T, \bot)$ for every transaction in h_p, if $T \leq^{h_p} T'$, then $T \leq^h T'$. Moreover, $\text{DEP}(h, \text{tr}(e), \bot) = \{e\} = \min_{\text{or}} \mathcal{P} \setminus h_p$. By 7.6 h is or-respectful, so for every T, $\min_{\text{or}} \text{DEP}(h, T, \bot) <_{\text{or}} e$; which implies $T <^h \text{tr}(e)$. By lemma 7.4, $\leq^h$ is a total order, so it coincides with $\leq^h$.
- Adding a `read`: As no transaction depends on $\text{tr}(e)$ and $\text{tr}(e) = \texttt{last}(h_p)$, if we prove that for every pair of transactions $\text{MINIMALDEPENDENCY}(h_p, T, T', \bot) = \text{MINIMALDEPENDENCY}(h, T, T', \bot)$, the lemma would hold. On one hand, $\text{DEP}(h, \text{tr}(e), \bot) = \text{DEP}(h_p, \text{tr}(e), \bot) = \text{tr}(e)$ and in the other hand, by lemma 7.6, $\min_{\text{or}} \text{DEP}(h_p, T, \bot) <_{\text{or}} \text{tr}(e)$. Finally, as $e \notin \text{DEP}(h, T, e')$, for every $T \neq \text{tr}(e), e' \neq \bot$, for every pair of transactions T, T', $\text{MINIMALDEPENDENCY}(h_p, T, T' \bot) = \text{MINIMALDEPENDENCY}(h, T, T' \bot)$.
- Swapping $r \in h$ and $w \in \text{tr}(e)$: As $\text{isSwappable}_{\mathcal{M}}(h \bullet e, r, w)$ is satisfied and h is or-respectful, for every event e' and transaction T, $\min_{\text{or}} \text{DEP}(h_p, T, e') = \min_{\text{or}} \text{DEP}(h, T, e')$, so for every pair of transactions $\text{MINIMALDEPENDENCY}(h_p, T, T', \bot) = \text{MINIMALDEPENDENCY}(h, T, T', \bot)$. In particular, this implies $T \leq^{h_p} T'$ if and only if $T \leq^h T'$ for every pair T, T' and $T \leq^h \text{tr}(r)$; so $\leq^h \equiv \leq_h$.

□

Proposition 7.7 is a very interesting result as it express the following fact: regardless of the computable path that leads to a history, the final order between events will be the same. This result will have a key role during both completeness and optimality, as it restricts the possible histories that precede another while describing the computable path leading to it. In addition, proposition 7.7 together with lemma 7.6 justify enlarging definition 7.5 with the canonical order instead the computable order; and it is this new shape the one we will be using during the rest of proof.

Lemma 7.8. *Any total history is or-respectful.*

Proof. Let h be a total history and T, T' a pair of transactions s.t. $T \leq_{\text{or}} T'$. If $T \leq^h T'$, then the statement is satisfied; so let's assume the contrary: $T' \leq^h T$. If T' $[\text{so} \cup \text{wr}]^* T$, then for every $e \in T, e' \in T'$ $\exists c \in h$ s.t. $\text{tr}(c) \leq_{\text{or}} \text{tr}(e)$, $\text{tr}(e')$ $[\text{so} \cup \text{wr}]^* \text{tr}(c)$, $\text{SWAPPED}(h, c)$ and $c \leq^h e$; so the property is satisfied. Otherwise, by definition of MINIMALDEPENDENCY, there exists $r' \in h$ s.t.

T' $[\mathsf{so} \cup \mathsf{wr}]^*$ $\mathsf{tr}(r')$ and $\mathsf{tr}(r') \leq_{\mathsf{or}} T$. Moreover, by CANONICALORDER's definition, $\mathsf{tr}(r) \leq^h T$. Finally SWAPPED(h, r') holds as it is the minimum element according or. To sum up, $R^{\mathsf{or}}(h)$ holds. □

Algorithm 4 PREV

```
1:  procedure PREV(h)
2:      if h = ∅ then
3:          return ∅
4:      end if
5:      a ← last(h)
6:      if ¬SWAPPED(h, a) then
7:          return h \ a
8:      else
9:          return MAXCOMPLETION(h \ a, {e | e ∉ (h \ a) ∧ e <_or h.wr(a)})
10:     end if
11: end procedure

12: procedure MAXCOMPLETION(h, D)
13:     if D ≠ ∅ then
14:         e ← min_{<_or} D
15:         if e.type() ≠ read then
16:             return MAXCOMPLETION(h • e, D \ {e})
17:         else
18:             let w s.t. ISMAXIMALLYADDED_M(h •_w e, e,)
19:             return MAXCOMPLETION(h •_w e, D \ {e})
20:         end if
21:     else
22:         return h
23:     end if
24: end procedure
```

Function 4 produce a history that are meant to be the previous step of a reachable history. Thanks to this definition, we will show that every total history has a computable path based on applying PREV^{-1} function iteratively until the objective history is reached.

TODO (somewhere before): if $h \longrightarrow$ SWAP$(h \bullet e, r, w)$ "in one step", actually from h we go to $h \bullet e$ and from it to the swapped.

LEMMA 7.9. *For every* or*-respectful history* h, PREV(h) *is also* or*-respectful.*

PROOF. Let suppose $h \neq \emptyset$, $h_p =$ PREV(h), $a = \mathtt{last}(h)$, $e \in \mathcal{P}$ and $e' \in h_p$ s.t. $e \leq_{\mathsf{or}} e'$. As $R^{\mathsf{or}}(h)$ is satisfied, either $e \leq^h e'$ or $\exists e'' \in h, \mathsf{tr}(e'') \leq_{\mathsf{or}} \mathsf{tr}(e)$, $e'' \leq^h e$, $\mathsf{tr}(e')$ $[\mathsf{so} \cup \mathsf{wr}]^* \mathsf{tr}(e'')$ and SWAPPED(h, e''). If $\neg$SWAPPED(h, a), $h_p = h \bullet a$; so if $e \leq^h e'$, $e \leq^{h_p} e'$ and if not, $e'' \in h_p$, so $R^{\mathsf{or}}(h_p)$ holds.

Otherwise, $\textsc{swapped}(h, a)$ and we distinguish between the sets e and e' belong to. Firstly, for every pair of events $\hat{e} \in h_p \setminus h$, $\hat{e}' \in \textsc{dep}(h, \mathrm{tr}(\hat{e}, \bot))$, we know that $\mathrm{tr}(\hat{e}) \leq_{\mathrm{or}} \mathrm{tr}(\hat{e}')$. Therefore, $\min_{<_{\mathrm{or}}} \textsc{dep}(h, \mathrm{tr}(\hat{e}, \bot)) = \mathtt{begin}(\mathrm{tr}(\hat{e}))$. In addition, by construction of $\textsc{prev}(h)$ and or-respectfulness of h, for every $\hat{e} \in h, e'' \in h$, $\min_{<_{\mathrm{or}}} \textsc{dep}(h_p, \mathrm{tr}(\hat{e}), e'') = \min_{<_{\mathrm{or}}} \textsc{dep}(h, \mathrm{tr}(\hat{e}), e'')$. Combining both results, if e' belong to h, either $e \leq^{h_p} e'$ or exists a $e'' \in h$ s.t. $e'' \leq^{h_p} e$ and witness $R^{\mathrm{or}}(h)$ for e, e' (regardless of e's belonging to h, $e'' \leq^{h_p} e$). On the contrary, as h_p has no pending transactions, if $e' \notin h$, $\neg(\mathrm{tr}(e')\ [\mathrm{so} \cup \mathrm{wr}]^*\ \mathrm{tr}(e))$, so regardless if $\mathrm{tr}(e)\ [\mathrm{so} \cup \mathrm{wr}]^* \mathrm{tr}(e')$, $e \leq^{h_p} e'$. To sum up, $R^{\mathrm{or}}(h_p)$ holds. □

Lemma 7.10. *For every consistent history* or*-respectful h, if* $\textsc{prev}(h)$ *is reachable, then h is also reachable.*

Proof. Let suppose $h \neq \emptyset$, $h_p = \textsc{prev}(h)$ and $a = \mathtt{last}(h)$. If $\neg\textsc{swapped}(h, a)$, let $h_n = h_p \bullet a$ if a is not a read, $h_n = h_p \bullet_{h.\mathrm{wr}(a)} a$ in the other case. Either way, h_n is always reachable and it coincides with h. Otherwise, a is a read event and it swapped; so let us call $w = h.\mathrm{wr}(a)$. Firstly, as $\textsc{swapped}(h, a)$, $a <_{\mathrm{or}} w$, and by lemma 7.6, $R^{\mathrm{or}}(h_p)$ holds, so $a <_{h_p} w$ does; which let us conclude $\textsc{compute}(h_p)$ will always return (a, w) as a possible swap pair. In addition, all transactions in h_p are non-pending, so in particular $\mathtt{last}(h_p)$ is an end event. If we call $h_s = \textsc{swap}(h_p, a, w)$, and $h_p \setminus h = h_p \setminus h_s$ would hold, as $h \subseteq h_p, h_s \subseteq h_p$, then $h = h_s$; which would allow us to conclude h is reachable from h_p.

On one hand, if $e \in h_p \setminus h$, $e \notin h$ and $e <_{\mathrm{or}} w$. In particular, $\neg(\mathrm{tr}(e)\ [\mathrm{so} \cup \mathrm{wr}]^*\ \mathrm{tr}(w))$. Moreover, if $e \leq_{\mathrm{or}} a$, by $R^{\mathrm{or}}(h)$, either $e \leq^h a$ or $\exists e'' \in h, e'' \leq_{\mathrm{or}} e$ s.t. $\mathrm{tr}(a)\ [\mathrm{so} \cup \mathrm{wr}]^* \mathrm{tr}(e'')$, $e'' \leq^h e$ and $\textsc{swapped}(h, e'')$; both impossible situations as $e \notin h$ and $a = \mathtt{last}(h)$; so $a \leq_{\mathrm{or}} e$. In other words, $e \in h_p \setminus h_s$.

On the other hand, $e \in h_p \setminus h_s$ if and only if $\neg(\mathrm{tr}(e)\ [\mathrm{so} \cup \mathrm{wr}]^*\ \mathrm{tr}(w))$ and $a <_{\mathrm{or}} e <_{\mathrm{or}} w$. If $e \in h$ then $e \leq^h a$, and as h is or-respectful and $a \leq_{\mathrm{or}} e$, we deduce there exists a $e'' \in h$ s.t. $\mathrm{tr}(e'') \leq_{\mathrm{or}} \mathrm{tr}(a)$, $\mathrm{tr}(e)\ [\mathrm{so} \cup \mathrm{wr}]^* \mathrm{tr}(e'')$ and $\textsc{swapped}(h, e'')$. Moreover, as $c \in h$, $c \in h_p$; but as $\textsc{swapped}(h_p, c)$ and $\textsc{isSwappable}_{\mathcal{M}}(h, a, w)$ hold, $c \in h_s$ and so e does. This result leads to a contradiction, so $e \notin h$; i.e. $e \in h_p \setminus h$. □

Corollary 7.11. *In a consistent* or*-respectful history h whose previous history is reachable, if its last event a is swapped, h coincides with* $\textsc{swap}(\textsc{prev}(h), a, h.\mathrm{wr}(a))$.

Proof. It comes straight away from the proof of lemma 7.10. □

Lemma 7.12. *For every non-empty consistent* or*-respectful history h,* $h_p = \textsc{prev}(h)$ *and* $a = \mathtt{last}(h)$*, if* $\textsc{swapped}(h, a)$ *then* $\{e \in h_p \mid \textsc{swapped}(h_p, e)\} = \{e \in h \mid \textsc{swapped}(h, e)\} \setminus \{a\}$*, otherwise* $h_p = h \setminus a$.

Proof. Let $a = \mathtt{last}(h)$ and $h' = h \setminus a$. If a is not swapped, then $h_p = h'$, so the lemma holds immediately. Otherwise, as $h_p = \textsc{maxCompletion}(h')$, we will show that every event not belonging to $h_p \setminus h'$ is not swapped by induction on every recursive call to maxCompletion. Let us call $D = \{e \mid e \notin h' \wedge e <_{\mathrm{or}}\}$. This set, intuitively, contain all the events that would have been deleted from a reachable history h to produce h_p. In this setting, let us call $h_{|D|} = h'$, $D_{|D|} = D$ and $D_k = D_{k+1} \setminus \{\min_{<_{\mathrm{or}}} D_{k+1}\}$, $e_k = \min_{<_{\mathrm{or}}} D_k$ for every $k, 0 \leq k < |D|$ (i.e. $D_k = D_{k+1} \setminus \{e_{k+1}\}$). We will prove the lemma by induction on $n = |D| - k$, constructing a collection of histories h_k, $0 \leq k < |D|$, such that each one is an extension of its predecessor with a non-swapped event.

The base case, $h_{|D|}$ is trivial as by its definition it corresponds with h'. Let's prove the inductive case: $\{e \mid \textsc{swapped}(h_{k+1}, e)\} = \{e \mid \textsc{swapped}(h', e)\}$. If e_{k+1} is not a `read` event, $h_k = h_{k+1} \bullet e_{k+1}$ and $\{e \mid \textsc{swapped}(h_k, e)\} = \{e \mid \textsc{swapped}(h', e)\}$; as only `read` events can be swapped. Otherwise, by the model's causal-extensibility there exists a `write` event f_{k+1} s.t. writes the same variable and $\textsc{isConsistent}_{\mathcal{M}}(h_{k+1} \bullet_{f_{k+1}} e_{k+1}) \wedge \mathrm{tr}(f_{k+1})\ [\mathrm{so} \cup \mathrm{wr}]^*\ \mathrm{tr}(e_{k+1})$ holds. $\{e \mid \textsc{swapped}(h_{k+1}, e)\} = \{e \mid \textsc{swapped}(h_{k+1} \bullet_{f_{k+1}} e_{k+1}, e)\}$ holds. Let $E_{k+1} = \{w \mid \textsc{isConsistent}_{\mathcal{M}}(h_{k+1} \bullet_w e_{k+1}) \wedge \{e \mid \textsc{swapped}(h_{k+1}, e)\} = \{e \mid \textsc{swapped}(h_{k+1} \bullet_w e_{k+1}, e)\}\}$ and $w_{k+1} = \max_{\leq^{h_{k+1}}} E_{k+1}$. This element is well defined as f_{k+1} belongs to E_{k+1}. Therefore, $h_k = h_{k+1} \bullet_{w_{k+1}} e_{k+1}$ is consistent and $\{e \mid \textsc{swapped}(h_k, e)\} = \{e \mid \textsc{swapped}(h', e)\}$. Moreover, let's remark that as w_{k+1} is the maximum write event according to $\leq_{h_{k+1}}$ s.t. $\textsc{isConsistent}_{\mathcal{M}}(h_k)$ and $\{e \mid \textsc{swapped}(h_k, e)\} = \{e \mid \textsc{swapped}(h', e)\}$ and $R^{\mathrm{or}}(h)$, it also satisfies $\textsc{isMaximallyAdded}_{\mathcal{M}}(h_k, e_{k+1}, w_{k+1})$. Altogether, we obtain $h_p = h_0$; which let us conclude $\{e \in h_p \mid \textsc{swapped}(h_p, e)\} = \{e \in h' \mid \textsc{swapped}(h', e)\} = \{e \in h \mid \textsc{swapped}(h, e)\} \setminus \{a\}$. □

Lemma 7.13. *For every history h there exists some $k_h \in \mathbb{N}$ such that $\textsc{prev}^{k_h}(h) = \emptyset$.*

Proof. This lemma is immediate consequence of lemma 7.12. Let us call $\xi(h) = |\{e \in h \mid \textsc{swapped}(h, e)\}$, the number of swapped events in h, and let us prove the lemma by induction on $(\xi(h), |h|)$. The base case, $\xi(h) = |h| = 0$ is trivial as h would be $\emptyset$; so let's assume that for every history h such that $\xi(h) < n$ or $\xi(h) = h \wedge |h| < m$ there exists such k_h. Let h then a history s.t. $\xi(h) = n$ and $|h| = m$. $h_p = \textsc{prev}(h)$. On one hand, if $h_p = h \setminus a$ then $\xi(x_p) = \xi(h)$ and $|h_p| = |h| - 1$. On the other hand, if $h_p \neq h \setminus a$, $\xi(h_p) = \xi(h) - 1$. In any case, by induction hypothesis on h_p, there exists an integer k_{h_p} such that $\textsc{prev}^{k_{h_p}}(h_p) = \emptyset$. Therefore, $k_h = k_{h_p} + 1$ satisfies $\textsc{prev}^{k_h}(h) = \emptyset$. □

Proposition 7.14. *For every consistent* or*-respectful history h exists $k \in \mathbb{N}$ and some sequence of* or*-respectful histories $\{h_n\}_{n=0}^{k}$, $h_0 = \emptyset$ and $h_k = h$ such that the algorithm will compute.*

Proof. Let h a history, k the minimum integer such that $\textsc{prev}^k(h) = \emptyset$, which exists thanks to lemma 7.13 and $C = \{\textsc{prev}^{k-n}(h)\}_{n=0}^{k}$ a set of indexed histories. By the collection's definition and lemma 7.9, $h_0 = \textsc{prev}^k(h) = \emptyset$, $h_k = \textsc{prev}^0(h) = h$ and $R^{\mathrm{or}}(h_n)$ for every $n \in \mathbb{N}$; so let us prove by induction on n that every history in C is reachable. The base case, h_0, is trivially achieved; as it is always reachable. In addition, by lemma 7.10, we know that if h_n is reachable, h_{n+1} is it too; which proves the inductive step. □

Theorem 7.15. *Algorithm 2 is complete.*

Proof. By lemma 7.8, any consistent total history is or-respectful. As a consequence of proposition 7.14, there exist a sequence of reachable histories which h belongs to; so in particular, h is reachable. □

7.3 Optimality

I have this new proof shorter than the initial idea but I do not know if it doesn't have typos. TODO: reread it!

TODO: proof isSwappable soundness

Theorem 7.16. *Algorithm 2 is optimal.*

Proof. As the model is causal-extensible, any algorithm weakly optimal is also optimal. Let us prove that for every reachable history there is only a computable path that leads to it from $\emptyset$. By lemma 7.7, we know that for every reachable history h, $\leq_h \equiv \leq^h$. However, $\leq^h$ is an order that does not depend on the computable path that leads to h; so neither does $\leq_h$. Let's suppose that there exist two reachable histories h_1, h_2 s.t. $h_1 = \textsc{prev}(h)$ and that in one step of computation produce a common history h. If we prove they are identical, the algorithm would be optimal. Firstly, if $\texttt{last}(h)$ is not a swapped `read` event, by the definition of next function $h_2 = h \setminus \texttt{last}(h) = h_1$. Therefore, let us call $r = \texttt{last}(h)$ and $w = h.\mathsf{wr}(r)$. Because $\textsc{swapped}(h, r)$ holds, from h_2 to h it has to have happened a swap between r and w. But by corollary 7.11, $h = \textsc{swap}(h_1, r, w)$, so $h_1 \restriction_{h \setminus r} = h_2 \restriction_{h \setminus r}$. As $\textsc{isSwappable}_{\mathcal{M}}(h_i, r, w)$ holds and h_i is or-respectful (for $i \in \{1, 2\}$), $h_1 \setminus h = h_2 \setminus h$, and for every `read` event e, $h_1.\mathsf{wr}(e) = h_2.\mathsf{wr}(e)$. Therefore, $h_1 = h_2$. □

ACKNOWLEDGMENTS