

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 Polytechnique & 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), 20 pages.
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

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 Polytechnique & CNRS, France, cenea@irif.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.

2475-1421/2022/6-ART \$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

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*, $\text{var} : \mathcal{E} \rightarrow \mathcal{V}$, which if e 's type is write or read returns the variable it writes/reads in the database and another function called *value*, $\text{val} : \mathcal{E} \rightarrow \mathbb{N}$, that for every write w event returns the value that $\text{var}(w)$ will store after its execution. Note that for simplicity we assume $\text{val}(\mathcal{E}) \subseteq \mathbb{N}$, 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 “ $a \leftarrow \text{read}(x)$ ” or “ $\text{write}(x, 0)$ ” 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 $\min_{\text{po}_T} T$ has type begin, $\max_{\text{po}_T} T$ 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 $\text{tr} : \mathcal{E} \rightarrow \mathcal{T}$ 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 $\text{th}_{\mathcal{P}} : \mathcal{E} \rightarrow \mathbb{Z}_n$.

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* $h = \langle T, \text{so}, \text{wr} \rangle$ 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 $\text{wr} \subseteq \text{writes}(E) \times \text{reads}(E)$ 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 ,
- $\text{so} \cup \text{wr}$ is acyclic.

We denote by \emptyset the history with no vertices and we also write, with an abuse of notation, $T \text{ [wr]} T'$ whenever $\exists w, r \in T \times T'$ such that $w \text{ [wr]} r$. 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 $T \in \mathcal{T}$.

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.

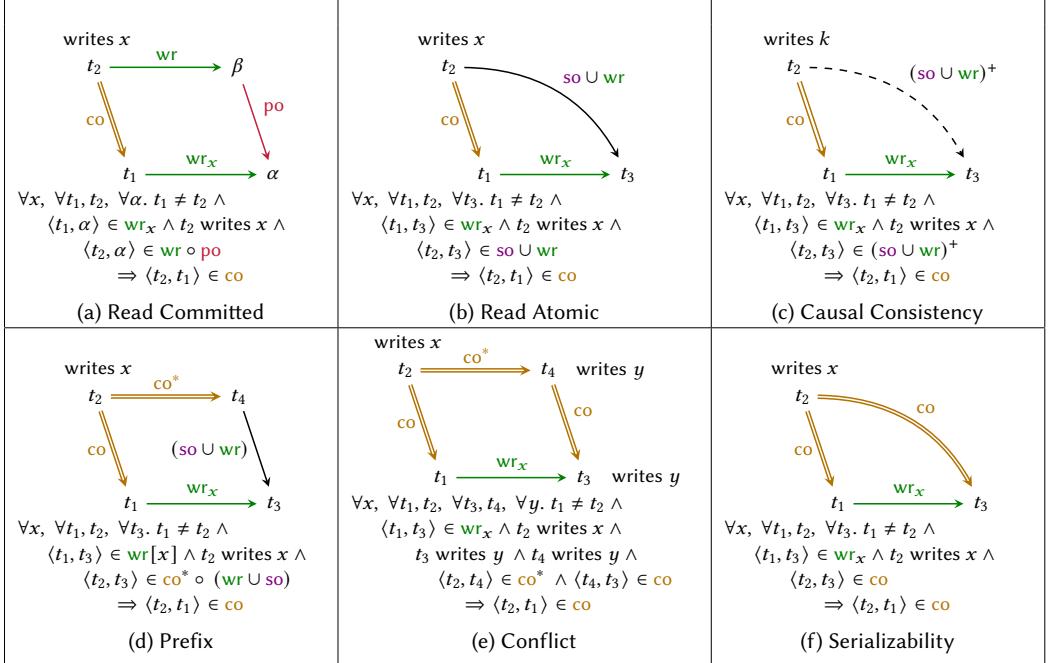


Fig. 1. Axioms defining isolations levels. The reflexive and transitive, resp., transitive, closure of a relation rel is denoted by rel^* , resp., rel^+ . Also, \circ denotes the composition of two relations, i.e., $rel_1 \circ rel_2 = \{(a, b) | \exists c. (a, c) \in rel_1 \wedge (c, b) \in rel_2\}$.

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 $so \cup wr$ 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 $so \cup wr$ is acyclic exists a commit order for those isolation levels.

3 MAXIMAL CLOSED MODELS

EXTENSIONS AND • OPERATOR NOT DEFINED IN THIS SECTION!!!!!!

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.

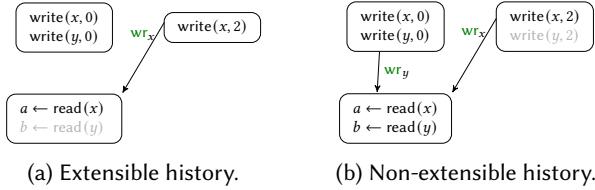


Fig. 2. Example of a dead-lock after swapping two events.

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 $r_1 = b \leftarrow \text{read}(y)$ and the wr-edge w_1 [wr] r_1 , where $w_1 = \text{write}(x, 0)$. However, this is not the case of h_2 : the only event that could be added in it is $w_2 = \text{write}(y, 2)$. If w_2 would be added in h_2 , any relation extending $\text{so} \cup \text{wr}$ 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 , $\text{tr}(r)$ is $\text{so} \cup \text{wr}$ -maximal while in h_2 $\text{tr}(w)$ is not. As real database executions forbid transactions reading from non-committed ones, it is reasonable to allow those transactions $\text{so} \cup \text{wr}$ -maximal to be executed completed without hindering the previous committed transactions.

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

- **Prefix-closedness:** Every $\text{so} \cup \text{wr}$ -prefix-closed sub-history of a consistent history is also consistent.
- **Maximal-extensibility:** Every non-total consistent history h can be consistently extended by executing an event from a $\text{so} \cup \text{wr}$ -maximal pending transaction T in h .
- **Past-readability:** For every history h and every $\text{so} \cup \text{wr}$ -maximal read event r there exists a write event w s.t. $h \bullet_w r$ is consistent and r is not-swapped **NEW CONSTRAINT ADDED!! Swapped not defined until a bit later!!** It is added as a constraint because it is stronger than maximal extensibility and only added in some small part of the proof (prev's correctness lemma). Nevertheless, it is an strengthening of maximal-extensibility; maybe it has to be adapted.

Comparing to the model requirements described in [cite viktor's algorithm](#), it is maximal-extensibility property the most weakened one. However, this weak formulation still forbids some axiomatic models such as Serializability [cite constantin's paper \(SER\)](#) to not be a maximal closed model [Appendix cite](#).

THEOREM 3.2. *Causal Consistency (CC), Read Atomic (RA) and Read Committed (RC) are maximal closed models.*

PROOF.

Prefix-closedness: Let h be a consistent history. As any $\text{so} \cup \text{wr}$ -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 it.

Maximal-extensibility: Let h a non-total consistent history with a $\text{so} \cup \text{wr}$ -maximal pending transaction T . Let $e = \min_{\text{po}_T} \{e' \in T \setminus h\}$. We proceed analyzing case by case depending on e 's type:

- begin case. This case is impossible as T was a pending transaction in h .
- end case. As $h \bullet e$ is a complete history with the same set of relations, it is consistent.
- write case. Let $h' = h \bullet e$. In this history there is no read event r such that $e \text{ [wr]} r$, as all those edges are already fixed in h . Moreover, as T is $\text{so} \cup \text{wr}$ -maximal, there is no transaction T' such that $T \text{ [so} \cup \text{wr]} T'$; and therefore, it can never take the role of t_1 nor t_2 in the axioms. To sum up, h' correspond to the same graph as h ; so it is consistent.
- read case. In this case, we have to prove that there exists a write event w such that $h'_w = h \bullet_w e$ is consistent. In particular, if $x = \text{var}(e)$, we will show it by induction on the number of cycles imposed by history h'_w to any total order that satisfies the axioms; for every write event w with same variable such that $\forall y \in \mathcal{V}, T' \in h$ s.t. $T' \text{ [wr}_y] \text{ tr}(w)$ either $\neg(T' \text{ [wr} \cup \text{so} \cup \text{co}]^* \text{ tr}(w))$ or $\neg(\text{tr}(w) \text{ writes } y)$. Let's remark that as the initial write of variable x satisfies this property, we know this set is nonempty and to prove the theorem it suffices to prove the commit relation deduced from $\text{so} \cup \text{wr}_w$, co_w , is acyclic.
 - Base case: There is no cycles in h_w . As co_w is acyclic, h_w is consistent under CC, RA and RC.
 - Inductive case: The theorem is true if h_w has at most n $\text{so} \cup \text{wr}_w \cup \text{co}_w$ cycles. Let's suppose that the history h_w has $n+1$ cycles of this kind and let's see how to obtain a history h'_w with one less cycle. As for every variable y and every transaction T' s.t. $T' \text{ [wr}_y] T$ either $\neg(T' \text{ [wr} \cup \text{so} \cup \text{co}]^* \text{ tr}(w))$ or $\neg(\text{tr}(w) \text{ writes } y)$, if there is a cycle in h_w is because there is a transaction T'' such that T'' writes x , $\text{tr}(w) \text{ [wr} \cup \text{so} \cup \text{co}]^* T''$ and $T'' \text{ [wr} \cup \text{so}]^+ \text{ tr}(e)$ (CC), $T'' \text{ [wr} \cup \text{so}] \text{ tr}(e)$ (RA) or $T'' \text{ [wr}] \text{ tr}(e)$ (RC) (figure 1).

Let w' a write event in T'' s.t. $\text{var}(w') = x$ and $h'_{w'} = h \bullet_w e$. First, as h is consistent and $\text{tr}(e)$ depends on T'' , it is clear that the cycle between $\text{tr}(w)$ and T'' has disappeared. Adding that T is $\text{so} \cup \text{wr}$ -maximal, we deduce that any cycle in h_w is due to the co_w -edges forced by adding $\text{tr}(w) \text{ [wr}] \text{ tr}(e)$ in h . Therefore, it can be checked case by case that for any cycle in h_w there is a cycle in h_w (starting from a graph with a cycle between T' , $\text{tr}(w)$ and T plus a cycle between T'' , $\text{tr}(w)$ and T in h_w we can deduce there is a cycle between T'' , T' and T in h_w). **there is no space no time to draw all four graphs, and do the reasoning (which is quite simple and in all cases is the same).** Moreover, as h is consistent, there cannot be any co -edge created with T' as t_2 and T as t_3 that was not already in h , so the event w' holds that for every variable y and every transaction T' s.t. $T' \text{ [wr}_y] T$ either $\neg(T' \text{ [wr} \cup \text{so} \cup \text{co}]^* \text{ tr}(w'))$ or $\neg(\text{tr}(w') \text{ writes } y)$; i.e. w' satisfies the induction hypothesis. To sum up, h_w has n cycles, so by induction hypothesis there is an acyclic history h'_w ; i.e. h can be maximally extended.

□

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, **New version! (problems in last section!)**
- 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 O_M^n is defined as the minimal collection containing all algorithms M -sound, complete and optimal that employ polynomial memory and allows at most n pending transactions.

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

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 N, E, C, P, S to be defined. Thanks to this approach we will show the existence of some algorithms and easily prove some of their properties.

Algorithm 1 EXPLORE algorithm

Input: h : history

```

1:  $e \leftarrow N(h)$ 
2: if  $TYPE(e) = \perp$  then
3:   if  $E(h)$  then
4:     output  $h$ 
5:   end if
6:   return
7: else if  $TYPE(e) = \text{read}$  then
8:    $x \in \mathcal{V}$  s.t.  $e$  reads  $x$ 
9:   for all  $w \in h$  s.t.  $w$  writes  $x$  do
10:    EXPLORE( $h \bullet_w e$ )
11: else
12:   EXPLORE( $h \bullet e$ )
13: end if
14:  $l \leftarrow C(h)$ 
15: for all  $(\alpha, \beta) \in l$  do
16:   if  $P(h, \alpha, \beta)$  then
17:     EXPLORE( $S(h, \alpha, \beta)$ )
18:   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 be added h along with an eventual *wr-edge*. Moreover, at some point during the search traversal determined by C the algorithm will *compute* some collection of events α, β that may need 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, C, P, S); where N, E, C, P, S are functions defined as follows:

- N : $\mathcal{H}_\mathcal{P} \rightarrow \mathcal{E}_\mathcal{P}$, where for every $h \in \mathcal{H}_\mathcal{P}$, $N(h) \notin h$ and for every event e s.t. $e \text{ [so]} N(h)$, $e \in h$,
- E : $\mathcal{H}_\mathcal{P} \rightarrow \{0, 1\}$, that determines if an execution is total or not,
- C : $\mathcal{H}_\mathcal{P}^< \rightarrow \mathcal{E}_\mathcal{P}^* \times \mathcal{E}_\mathcal{P}^*$, where for every history h , $C(h) = (\alpha, \beta)$, $\alpha \cap \beta = \emptyset$ and for every $(e, e') \in \alpha \times \beta$, $e <_h e'$.
- P : $\mathcal{H}_\mathcal{P}^< \times \mathcal{E}_\mathcal{P}^* \times \mathcal{E}_\mathcal{P}^* \rightarrow \{0, 1\}$,
- S : $\mathcal{H}_\mathcal{P}^< \times \mathcal{E}_\mathcal{P}^* \times \mathcal{E}_\mathcal{P}^* \rightarrow \mathcal{H}_\mathcal{P}^<$, where for every h, α, β , we have $S(h, \alpha, \beta) = h'$, $\alpha, \beta \in h'$, for every $(e, e') \in \alpha \times \beta \Rightarrow e >_{h'} e'$ and there exists some $(e, e') \in (\alpha, \beta)$ s.t. $\text{tr}(e') \text{ [wr]} \text{tr}(e)$.

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 STATELESS DPOR ALGORITHMS FOR MAXIMAL-EXTENSIBLE MODELS

The main goal in this work is describing a deterministic algorithm for transactional model-checking under a [maximal-extensible model?](#) \mathcal{M} ; obtaining all possible behaviors a program may have. We present during this section a sound, complete and optimal algorithm employing polynomial memory, i.e. a member of $\mathcal{S}_\mathcal{M} \cap \mathcal{O}_\mathcal{M}^n$ for some $n \in \mathbb{N}$. 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 simplicity, we will omit proof of its properties as a later refinement found in section ?? will be the one proved.](#)

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

In addition, we assume the existence of a total order between the events in every history, called *history order* and denoted as $<_h$, 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:

$$\text{NEXT}(h) = \begin{cases} \min_{\text{or}} \{e \in \mathcal{E} \mid e \notin h\} & \text{if } \#\text{PENDING}_h(T) \\ \min_{\text{or}} \{e \in \mathcal{E} \setminus h \mid e \in \text{PENDING}_h(T)\} & \text{otherwise} \end{cases}$$

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. If the event $e = \text{NEXT}(h)$ is begin, write or end, we will denote by $h \bullet e$ the history $h' = \langle E', \text{so}', \text{wr} \rangle$ where $E' = \text{events}(h) \cup \{e\}$ and $\text{so}' = \text{so} \cup \{(e', e) \mid e' \in h \wedge \text{th}(e) = \text{th}(e')\}$. 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 $\langle E', \text{so}', \text{wr} \cup \{(w, r)\} \rangle$, where E' and so' defined as before. It has to be moved to the definitions' section.

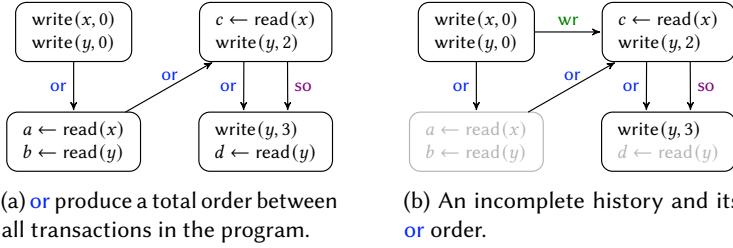


Fig. 3. Some possible oracle order between transactions.

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 .

5.1 Extending 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 pieces 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 $h'_w = h \bullet_w e$, $w \in \text{write}(h)$. 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).

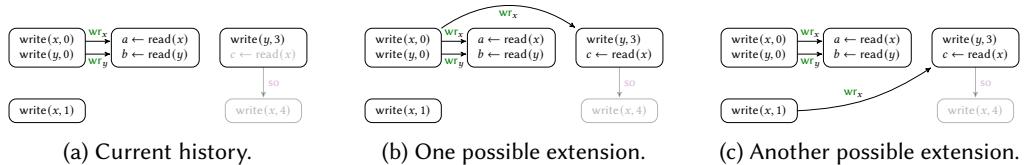


Fig. 4. Extensions of a history by adding a read event.

Conversely, adding a write w is more complicated; multiple events may read this new event. The number of possible histories is exponential, $2^{|\text{read}(h)|}$. Here, instead of generating every single one of them and check afterwards if they are consistent, we prefer to generate a small subset and only extends those ones that are consistent; by the prefix-closedness of our model, an inconsistent history only leads to inconsistent ones.

Our protocol is defined as follows: we will select one read event that will be the first event in h reading from w while the ones that proceed r will be marked as postponed; they will have to be

re-executed. As in a more formal way definition 5.1 states, This read r would be thereafter called *swapped*:

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

- For $w = h.\text{wr}(r)$, $w <_h r$ and $w >_{\text{or}} r$.
- There is no transaction besides $\text{tr}(r)$ that depends on $\text{tr}(w)$: $\nexists T <_{\text{or}} \text{tr}(r)$ such that $T <_h \text{tr}(r)$ and $\text{tr}(w) [\text{so} \cup \text{wr}]^+ T$. Note: I changed $T \neq \text{tr}(r)$ to $T <_{\text{or}} \text{tr}(r)$ for the proof as by or-respectfulness (defined in proof's section), we obtain both are equivalent.
- r is the first read that reads from $\text{tr}(w)$: $\nexists r' \in \text{tr}(r), r' <_h r$ such that $\text{tr}(h.\text{wr}(r')) = \text{tr}(w)$

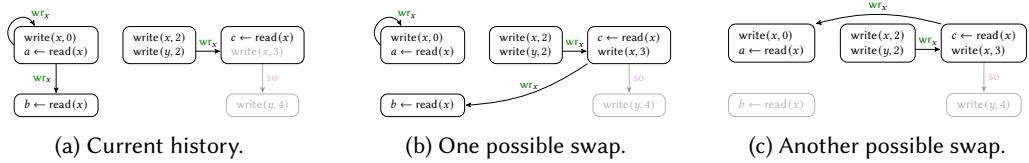


Fig. 5. Extensions of a history by adding a write event.

Let us show with an example the power of swapping. Here, in Figure 5a we can see an incomplete history whose NEXT event, $w := \text{write}(x, 3)$, is a write event. There are four possible histories, depending if the first and second read events, $r_a := a \leftarrow \text{read}(x)$ and $r_b := b \leftarrow \text{read}(x)$, read or not from w ($c \leftarrow \text{read}(x)$ 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 $w [\text{wr}] r_b$ in h_b . The other two cases that modify the write-read relation are due to the edge $w [\text{wr}] r_a$; 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, by the read-rule if $w [\text{wr}] r_b$ 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 $\text{tr}(w)$ does not depend on. Otherwise, deleting some event e such that $\text{tr}(e) [\text{so} \cup \text{wr}]^* \text{tr}(w)$ 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.

5.2 Avoiding inconsistent branches

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

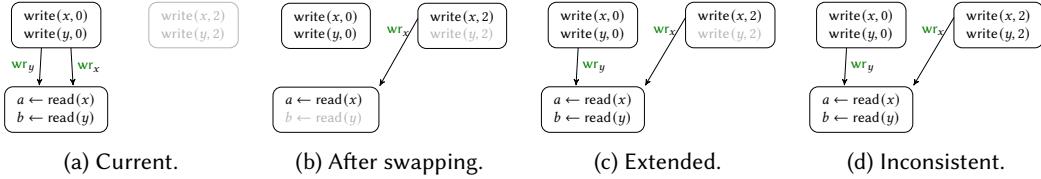


Fig. 6. Example of a dead-lock after swapping two events.

One example of this undesired behavior can be easily seen under RA memory model with a program like the one 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 $w_x := \text{write}(x, 2)$, one possible action would be swapping w_x with the event $a \leftarrow \text{read}(x)$; obtaining the history h_2 portrayed in Figure 6b. By definition of the NEXT function, that history shall be extended with $r_b := b \leftarrow \text{read}(y)$, 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 $\text{so} \cup \text{wr}$ -maximal. As NEXT function always prioritize pending transactions, for never get out from theorem 3.2 hypothesis we have to not produce pending transactions that are non $\text{so} \cup \text{wr}$ -maximal. A simple solution is always executing histories in isolation, i.e. having exactly one pending transaction. Then, this transaction will $\text{so} \cup \text{wr}$ -maximal, otherwise there would be a previous point where two pending transactions coexisted. Hence, we are not going to swap just after executing a write event but when its transaction is completed.

Algorithm 2 Recursive STMC

```

Input:  $h$ : history.
1: if  $\neg \text{ISCONSISTENT}_{\mathcal{M}}(h)$  then return
2: else if  $\text{ISCOMPLETE}(h)$  then
3:    $\text{PROCESSHISTORY}(h)$ 
4: else
5:    $a \leftarrow \text{NEXT}(h)$ 
6:   switch  $a.\text{type}()$  do
7:     case begin :                                 $\triangleright$  begin and write cases coincide.
8:       case write :
9:          $\text{STMC}_{\text{rec}}(h \bullet a)$ 
10:        break
11:        case read :
12:           $x \in \mathcal{V}$  s.t.  $a$  reads  $x$ 
13:          for all  $w \in h$  s.t.  $w$  writes  $x$  do
14:             $h' \leftarrow h \bullet_w a$ 
15:             $\text{STMC}_{\text{rec}}(h')$ 
16:            break
17:        case end :
18:           $\text{STMC}_{\text{rec}}(h \bullet a)$ 
19:          for all  $w \in \text{tr}(a), x \in \mathcal{V}$  s.t.  $w$  writes  $x$  do
20:            for all  $r \in h$  s.t.  $r$  reads  $x$  do
21:               $D \leftarrow \{e \mid r <_h e \wedge \neg(\text{tr}(e) [\text{so} \cup \text{wr}]^* \text{tr}(a))\}$ 
22:               $h' \leftarrow (h \setminus D) \bullet a; h'.\text{wr}[r] \leftarrow w$ 
23:               $\text{STMC}_{\text{rec}}(h')$ 
24:            break
25: end if

```

Algorithm 2 is in charge of extending non-total consistent histories, detecting inconsistencies as soon as they are created (line 1). For doing so, given a history h , it computes the NEXT event a with which h is going to be extended (line 5). If a has type begin or write there is only way to extend h , via $h \bullet a$ (line 8). In case a is a read event (line 11), we explore all possible histories h' where $h'.\text{wr}(a) = w$ (line 13). Otherwise, a has type end and we explore the case where no swap is produced (line 18) and when it is produced (line 20). There we compute the set D of events that have to be deleted from the history as explained in 5.1, removing it from h and setting a new wr edge between r and w .

Only in the last case there is one transaction ending up pending, but thanks to theorem 3.2, if h is executed in isolation, h' will also be. Both arguments justify soundness and non-blocking properties of our algorithm. Completeness, on the other hand, it is not a immediate property.

6 OPTIMAL STMC

In this section we present a refinement of algorithm 2 introducing a new property: *optimality*; which in this context optimality refers to not obtaining the same history twice. When extending a history h with an event e , if e is either a begin or write there is no possible redundancy as there can be only one extension, $h \bullet e$. If e is a read event, for every write event w we explore every

history $h'_w = h \bullet_w e$. As $h'_w.\text{wr}(e) \neq h'_{w'}.\text{wr}(e)$ if $w \neq w'$, adding a read event is no source of redundancy. This is not the case when e is an end-event and we produce a swap.

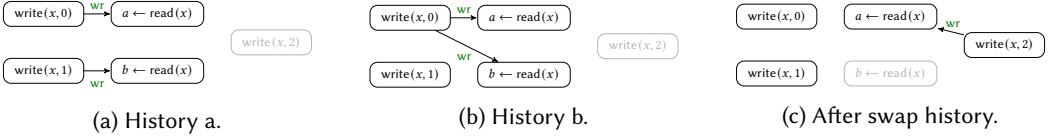


Fig. 7. Two different histories after swapping lead to a common one.

Let's suppose we have a program \mathcal{P} as depicted in figure 7, assuming $<_{\text{or}}$ order transactions as presented from left to right, top to bottom. Given \mathcal{P} , algorithm 2 computes the histories 7a, h , and 7b, h' . After adding the last transaction in both h, h' we produce a swap between $r_a := a \leftarrow \text{read}(x)$ and $w_2 := \text{write}(x, 2)$; deleting the event $r_b := b \leftarrow \text{read}(x)$ as $\neg(\text{tr}(r_b) [\text{wr} \cup \text{so}]^* \text{tr}(w_2))$. 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 limit. As the example in figure 7 shows, the key of redundancy lies in every **wr** edge that is going to be modified: if two histories only differ on those, the resultant history is the same. We define the following concept for controlling this situation:

Definition 6.1. An event e is *maximally added* in a history h if one of the following condition holds:

- (1) e is not a read event.
- (2) e is a non-swapped read event that reads the last **write** w event before e such that $h.\text{wr}(e) = w$ is consistent. **TODO: IMA = IsMaximallyAdded (doesnt fit yet (not even with IMA))**

$$\begin{aligned}
 \text{IMA}(h, e) = \exists x \in \mathcal{V}, \exists w \in \text{writes}(h) \text{ s.t. } & \quad w [\text{wr}_x] e \wedge \text{isConsistent}_{\mathcal{M}}(h) \\
 & \quad \wedge \\
 & \quad \forall w' \in \text{writes}(x), w' <_h e \Rightarrow \\
 & \quad (w' \leq_h w \vee \neg \text{isConsistent}_{\mathcal{M}}(h')) \\
 & \quad \left(\text{where } h' = h \wedge \left\{ \begin{array}{ll} h'.\text{wr}(e') = h.\text{wr}(e') & \text{if } e' \neq e \\ h'.\text{wr}(e) = w & \text{otherwise} \end{array} \right. \right)
 \end{aligned}$$

Intuitively, definition 6.1 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 establish 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 **isSWAPPABLE** and it will play the role of **P** function in our algorithm 1's instance.

$$\text{isSWAPPABLE}(h, r, w) = r \in \text{Del} \wedge \forall e \in \text{Del} : \text{IsMAXIMALLYADDED}(h, e) \quad \text{where } \text{Del} = \{e \mid r \leq_h e \wedge \neg(\text{tr}(e) [\text{wr} \cup \text{so}]^* \text{tr}(w))\}$$

Algorithm 3 Optimal recursive STMC

```

...
17: case end :
18:   STMCrec( $h \bullet a$ )
19:   for all  $w \in \text{tr}(a), x \in \mathcal{V}$  s.t.  $w$  writes  $x$  do
20:     for all  $r \in h$  s.t.  $r$  reads  $x$  do
21:        $D \leftarrow \{e \mid r <_h e \wedge \neg(\text{tr}(e) [\text{wr} \cup \text{so}]^* \text{tr}(a))\}$ 
22:       if  $\forall e \in D \cup \{r\}$ : IsMAXIMALLYADDED( $e$ ) then
23:          $h' \leftarrow (h \setminus D) \bullet a$ ;  $h'.\text{wr}[r] \leftarrow w$ 
24:         STMCrec( $h'$ )
25:       end if
26:   break
...

```

In algorithm 3 is depicted the slight modification needed in 2 for guaranteeing optimality: just adding the condition at line 22, detecting if every deleted event is maximally added, is enough to guarantee optimality.

Add at the end (here) the code of the final algorithm!!!!

7 WEAK DPOR ALGORITHMS FOR SNAPSHOT ISOLATION AND SERIALIZABILITY

As show in section PROOFS SECTION, algorithm CITE CC'S ALGORITHM's completeness proof is model-depending. Immediately, the question of its extensibility to other isolation levels arise. For understanding the difference between the formers and SI or SER, let's analyze how algorithm CITE CC'S ALGORITHM, denoted A thereafter I don't like calling A here, but i dont know what to do..., 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.

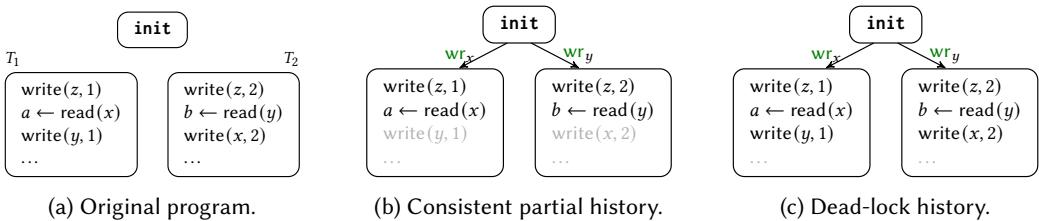


Fig. 8. A program and some partial histories. In gray, instructions not yet executed.

As \mathcal{P} is symmetric and every partial execution computed by A 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 $r_2 := b \leftarrow \text{read}(x)$ 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 A would have two pending transaction. To sum up, algorithm CITE CC'S ALGORITHM 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 constraints shall we remove in it to remain \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 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 write($y, 1$) or write($x, 2$) in it. Without loss of generality, let's assume $w_2 := \text{write}(x, 2) \notin h$. Shall I develop this a bit more? how?. As A is in $\mathcal{S}_{\mathcal{M}}$, for every $a = N(h)$, $a \notin h$; and as stated in lines 10, 12, the history $h \bullet a$ will always be explored. Therefore, as A

Moreover, as A is \mathcal{M} -sound, $N(h')$ cannot return w_2 as $\neg \text{IsConsistent}_{\mathcal{M}}(h' \bullet w_2)$, for any history h' extending h . Therefore, there is an extension h' of h where all events but those in T_2 po after w_x will be executed. In conclusion, $N(h') = \perp$ but h' is not total, so it must be a blocking execution. That contradicts A is an optimal algorithm.

THEOREM 7.1. *There is no algorithm in $\mathcal{S}_{\mathcal{M}} \cap \mathcal{O}_{\mathcal{M}}^n$, n in \mathbb{N} , for neither SI nor SER.*

PROOF. During this proof let's call \mathcal{P} to the program depicted in 8a. Let's suppose on the contrary that such algorithm A exists. 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. In particular, a consistent extension of the history shown in 8b will be computed with either write($y, 1$) or write($x, 2$) in it. Without loss of generality, let's assume $w_x := \text{write}(x, 2) \notin h$. Shall I develop this a bit more? how?. As A is weakly optimal, for every $a = N(h)$, $a \notin h$ (otherwise A will compute h and $h \bullet a = h$); and as stated in lines 10, 12, the history $h \bullet a$ will always be explored. Moreover, as A is \mathcal{M} -sound, $N(h')$ cannot return w_x as $\neg \text{IsConsistent}_{\mathcal{M}}(h' \bullet w_x)$, for any history h' extending h . Therefore, there is an extension h' of h where all events but those in T_2 po after w_x will be executed. In conclusion, $N(h') = \perp$ but h' is not total, so it must be a blocking execution. That contradicts A is an optimal algorithm.

The first case is due to the non-wr U so-maximality of SI, SER (second example we discussed). The second one is the example when there is only one branch computed (first example we discussed). \square

The reason why theorem 7.1 holds for SI and SER but not for the rest of isolation levels described in 1 is due to the maximal-extensibility of the rest of the models. Under any of them the event write($x, 2$) would always be eligible to be added, and therefore any final dead-lock would be reached.

THEOREM 7.2. *There exists an algorithm in $\mathcal{S}_{\mathcal{M}} \cap \mathcal{W}_{\mathcal{M}}$ for $\mathcal{M} \in \{\text{SI}, \text{SER}\}$.*

8 PROOF OF THE ALGORITHMS

Introduction

8.1 Soundness

LEMMA 8.1. *Let h a consistent incomplete history and s a state of the program such that h appears at line ???. Let h' the next history computed by the algorithm from s . Then $\text{PREV}(h') = h$.*

8.2 Completeness

Problem of this proofs: I assume optimality before proving it! ~~I don't know what wording we shall apply (I tried to modify it, let's see if it is noticeable).~~

Algorithm 4 CANONICAL ORDER

```

1: procedure CANONICALORDER( $h, T, T'$ )
2:   return  $T [so \cup wr]^* T' \vee$ 
3:    $(\neg(\text{tr}(T')) [so \cup wr]^* T) \wedge \text{MINIMALDEPENDENCY}(h, T, T', \perp)$ 
4: end procedure

5: procedure MINIMALDEPENDENCY( $h, T, T', e$ )
6:   let  $a = \min_{<_{\text{or}}} \text{DEP}(h, T, e); a' = \min_{<_{\text{or}}} \text{DEP}(h, T', e)$ 
7:   if  $a \neq a'$  then
8:     return  $a <_{\text{or}} a'$ 
9:   else
10:    return MINIMALDEPENDENCY( $h, T, T', a$ )
11:   end if
12: end procedure

13: procedure DEP( $h, T, e$ )
14:   return  $\{r \mid \exists w \text{ s.t. } T [so \cup wr]^* \text{tr}(w) \wedge w [wr] r \wedge \text{tr}(r) [so \cup wr]^+ \text{tr}(e)\} \cup T$ 
15: end procedure

```

The function CANONICALORDER produces a relation between transactions in a history, denoted \leq^h . In algorithm 4's description, we denote \perp as the end of the program, which always exists, and that is **so**-related with every single transaction.

LEMMA 8.2. *For every history h , event e and transaction T , $\text{DEP}(h, T, \min_{<_{\text{or}}} \text{DEP}(h, T, e)) \subseteq \text{DEP}(h, T, e)$. Moreover, if $\text{DEP}(h, T, e) \neq T$, the inclusion is strict.*

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

LEMMA 8.3. *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 8.2 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. $\min_{<_{\text{or}}} \text{DEP}(h, T, e) \neq \min_{<_{\text{or}}} \text{DEP}(h, T', e)$. \square

LEMMA 8.4. *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 $e \in h$, $\min_{\leq_{\text{or}} \text{DEP}}(a, e) \leq_{\text{or}} \min_{\leq_{\text{or}} \text{DEP}}(b, e)$, so $a <^h c$.
 - If $\neg(a [\text{so} \cup \text{wr}]^* b)$ but $b [\text{so} \cup \text{wr}]^* c$, then for every $e \in h$, $\min_{\leq_{\text{or}} \text{DEP}}(b, e) \leq_{\text{or}} \min_{\leq_{\text{or}} \text{DEP}}(c, e)$, 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 $\text{MINIMALDEPENDENCY}(h, a, b, \perp)$ and $\text{MINIMALDEPENDENCY}(h, b, a, \perp)$ 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 8.3) and $\neg\text{MINIMALDEPENDENCY}(h, a, b, e)$, then $\text{MINIMALDEPENDENCY}(h, b, a, e)$; so $b <^h a$.

□

Definition 8.5. 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 8.6. Every reachable history is or-respectful.

PROOF. We will prove it by induction on the number of code algorithm'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 $h' = \text{SWAP}(h, r, w)$ for some $r, w \in h$ s.t. $\text{ISSWAPPABLE}(h, r, w)$ 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 $\text{ISSWAPPABLE}(h, r, w)$ 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 $\text{ISSWAPPABLE}(h, r, w)$; so $r \leq_{\text{or}} a$. Taking $e'' = r$ the property is witnessed. \square

LEMMA 8.7. *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 [cite algorithm](#). 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 $\text{DEP}(h_p, T, \perp) = \text{DEP}(h, T, \perp)$ for every transaction in h_p , if $T \leq^{h_p} T'$, then $T \leq^h T'$. Moreover, $\text{DEP}(h, \text{tr}(e), \perp) = \{e\} = \min_{\text{or}} \mathcal{P} \setminus h_p$. By 8.6 h is **or**-respectful, so for every T , $\min_{\text{or}} \text{DEP}(h, T, \perp) <_{\text{or}} e$; which implies $T <^h \text{tr}(e)$. By lemma 8.4, \leq^h is a total order, so it coincides with \leq^h .
- Adding a read: As no transaction depends on $\text{tr}(e)$ and $\text{tr}(e) = \text{last}(h_p)$, if we prove that for every pair of transactions $\text{MINIMALDEPENDENCY}(h_p, T, T', \perp) = \text{MINIMALDEPENDENCY}(h, T, T', \perp)$, the lemma would hold. On one hand, $\text{DEP}(h, \text{tr}(e), \perp) = \text{DEP}(h_p, \text{tr}(e), \perp) = \text{tr}(e)$ and in the other hand, by lemma 8.6, $\min_{\text{or}} \text{DEP}(h_p, T, \perp) <_{\text{or}} \text{tr}(e)$. Finally, as $e \notin \text{DEP}(h, T, e')$, for every $T \neq \text{tr}(e), e' \neq \perp$, for every pair of transactions T, T' , $\text{MINIMALDEPENDENCY}(h_p, T, T', \perp) = \text{MINIMALDEPENDENCY}(h, T, T', \perp)$.
- Swapping $r \in h$ and $w \in \text{tr}(e)$: As $\text{ISSWAPPABLE}(h, r, w)$ 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 $\text{MINIMALDEPENDENCY}(h_p, T, T', \perp) = \text{MINIMALDEPENDENCY}(h, T, T', \perp)$. 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 $\leq^h \equiv \leq_h$.

\square

As $\leq^h \equiv \leq_h$ for any reachable history, we will extend $R^{\text{or}}(h)$ to any history changing \leq_h to \leq^h in 8.5 whenever it is needed. This property is not something reachable histories satisfy but also, as next lemma shows, total histories with \leq^h order do; which justify it as an useful tool for proving completeness.

LEMMA 8.8. *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. $T' [\text{so} \cup \text{wr}]^* \text{tr}(r')$ and $\text{tr}(r') \leq_{\text{or}} T$. Moreover, by **CANONICALORDER**'s definition, $\text{tr}(r) \leq^h T$. Finally $\text{SWAPPED}(h, r')$ holds as it is the minimum element according **or**. To sum up, $R^{\text{or}}(h)$ holds. \square

Algorithm 5 PREV

```

1: procedure PREV( $h$ )
2:   if  $h = \emptyset$  then
3:     return  $\emptyset$ 
4:   end if
5:    $a \leftarrow \text{last}(h)$ 
6:   if  $\neg \text{SWAPPED}(h, a)$  then
7:     return  $h \setminus a$ 
8:   else
9:     return MAXCOMPLETION( $h \setminus a, \{e \mid e \notin (h \setminus a) \wedge e <_{\text{or}} h.\text{wr}(a)\}$ )
10:  end if
11: end procedure

12: procedure MAXCOMPLETION( $h, D$ )
13:   if  $D \neq \emptyset$  then
14:      $e \leftarrow \min_{<_{\text{or}}} D$ 
15:     if  $e.\text{type}() \neq \text{read}$  then
16:       return MAXCOMPLETION( $h \bullet e, D \setminus \{e\}$ )
17:     else
18:       let  $w$  s.t. ISMAXIMALLYADDED( $h \bullet_w e, e$ )
19:       return MAXCOMPLETION( $h \bullet_w e, D \setminus \{e\}$ )
20:     end if
21:   else
22:     return  $h$ 
23:   end if
24: end procedure

```

LEMMA 8.9. *For every non-empty consistent or-respectful history h , $h_p = \text{PREV}(h)$ and $a = \text{last}(h)$, if $\text{SWAPPED}(h, a)$ then $\{e \in h_p \mid \text{SWAPPED}(h_p, e)\} = \{e \in h \mid \text{SWAPPED}(h, e)\} \setminus \{a\}$, otherwise $h_p = h \setminus a$.*

PROOF. Let $a = \text{last}(h)$ and $h' = h \setminus a$. If a is not swapped, then $h_p = h'$, so the lemma holds immediately. Otherwise, as $h_p = \text{MAXCOMPLETION}(h')$, 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 $D = \{e \mid e \notin h' \wedge e <_{\text{or}}\}$. 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 $D_k = D_{k+1} \setminus \{\min_{<_{\text{or}}} D_{k+1}\}$, $e_k = \min_{<_{\text{or}}} D_k$ 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: $\{e \mid \text{SWAPPED}(h_{k+1}, e)\} = \{e \mid \text{SWAPPED}(h', e)\}$. If e_{k+1} is not a read event, $h_k = h_{k+1} \bullet e_{k+1}$ and $\{e \mid \text{SWAPPED}(h_k, e)\} = \{e \mid \text{SWAPPED}(h', e)\}$; as only read events can be swapped. Otherwise, by the model's **past-readability** there exists a write event f_{k+1} s.t. writes the same variable and $\text{ISCONSISTENT}_{\mathcal{M}}(h_{k+1} \bullet_{f_{k+1}} e_{k+1}) \wedge \{e \mid \text{SWAPPED}(h_{k+1}, e)\} = \{e \mid \text{SWAPPED}(h_{k+1} \bullet_{f_{k+1}} e_{k+1}, e)\}$ holds. Let $w_{k+1} = \max_{\leq h_{k+1}} \{w \mid \text{ISCONSISTENT}_{\mathcal{M}}(h_{k+1} \bullet_w e_{k+1}) \wedge \{e \mid \text{SWAPPED}(h_{k+1}, e)\} = \{e \mid \text{SWAPPED}(h_{k+1} \bullet_w e_{k+1}, e)\}\}$. Therefore, $h_k = h_{k+1} \bullet_{w_{k+1}} e_{k+1}$ is consistent and $\{e \mid \text{SWAPPED}(h_k, e)\} = \{e \mid \text{SWAPPED}(h', e)\}$. Moreover, let's remark that as w_{k+1} is

the maximum write event according to $\leq_{h_{k+1}}$ s.t. $\text{ISCONSISTENT}_{\mathcal{M}}(h_k)$ and $\{e \mid \text{SWAPPED}(h_k, e)\} = \{e \mid \text{SWAPPED}(h', e)\}$ and $R^{\text{or}}(h)$, it also satisfies $\text{ISMAXIMALLYADDED}(h_k, e_{k+1}, w_{k+1})$. Altogether, we obtain $h_p = h_0$; which let us conclude $\{e \in h_p \mid \text{SWAPPED}(h_p, e)\} = \{e \in h' \mid \text{SWAPPED}(h', e)\} = \{e \in h \mid \text{SWAPPED}(h, e)\} \setminus \{a\}$. In orange where I use the new constraint. \square

LEMMA 8.10. *For every history h there exists some $k_h \in \mathbb{N}$ such that $\text{PREV}^{k_h}(h) = \emptyset$.*

PROOF. This lemma is immediate consequence of lemma 8.9. Let us call $\xi(h) = |\{e \in h \mid \text{SWAPPED}(h, e)\}|$, 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 $|h| = m$. $h_p = \text{PREV}(h)$. On one hand, if $h_p = h \setminus a$ then $\xi(h_p) = \xi(h)$ 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 $\text{PREV}^{k_{h_p}}(h_p) = \emptyset$. Therefore, $k_h = k_{h_p} + 1$ satisfies $\text{PREV}^{k_h}(h) = \emptyset$. \square

PROPOSITION 8.11. *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 $\text{PREV}^k(h) = \emptyset$, which exists thanks to lemma 8.10 and $C = \{\text{PREV}^{k-n}(h)\}_{n=0}^k$ a set of indexed histories. By the collection's definition and lemma ??, $h_0 = \text{PREV}^k(h) = \emptyset$, $h_k = \text{PREV}^0(h) = h$ and $R^{\text{or}}(h_n)$ 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 ??, we know that if h_n is reachable, h_{n+1} is it too; which proves the inductive step. \square

THEOREM 8.12. *Algorithm cite algorithm is complete.*

PROOF. By lemma 8.8, any consistent total history is or-respectful. As a consequence of proposition 8.11, there exist a sequence of reachable histories which h belongs to; so in particular, h is reachable. \square

ACKNOWLEDGMENTS