Dynamic Partial Order Reduction for Checking Correctness Against Weak Isolation Levels

ANONYMOUS AUTHOR(S)

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.

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.

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

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

$$x \in \mathsf{Vars} \quad a \in \mathsf{LVars}$$

$$\begin{array}{rlrl}
\mathsf{Prog} ::= & \mathsf{Sess} \;\mid\; \mathsf{Sess} \,||\, \mathsf{Prog} & \mathsf{Body} ::= & \mathsf{Instr} \;\mid\; \mathsf{Instr}; \mathsf{Body} \\
\mathsf{Sess} ::= & \mathsf{Trans} \;\mid\; \mathsf{Trans}; \mathsf{Sess} & \mathsf{Instr} ::= & \mathsf{InstrDB} \;\mid\; a := e \;\mid\; \mathtt{if}(\phi(\vec{a}))\{\mathsf{Instr}\} \\
\mathsf{Trans} ::= & \mathtt{begin}; \mathsf{Body}; \mathtt{commit} & \mathsf{InstrDB} ::= & a := \mathtt{read}(x) \;\mid\; \mathtt{write}(x, a)
\end{array}$$

Fig. 1. Program syntax. The set of global variables is denoted by Vars while LVars denotes the set of local variables. We use ϕ to denote Boolean expressions, and e to denote expressions over local variables interpreted as values. We use $\vec{\cdot}$ to denote vectors of elements.

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 TRANSACTIONAL PROGRAMS

2.1 Program Syntax

Figure 1 lists the definition of a simple programming language that we use to represent applications running on top of a database. A program is a set of *sessions* running in parallel, each session being composed of a sequence of *transactions*. Each transaction is delimited by `begin` and `commit` instructions, and its body contains instructions that access the database and manipulate a set LVars of local variables.[1] We use symbols a, b, etc. to denote elements of LVars.

For simplicity, we abstract the database state as a valuation to a set Vars of *global* variables[2]. Therefore, the instructions accessing the database correspond to reading the value of a global

[1]For simplicity, we assume that all the transactions in the program commit. Aborted transactions can be ignored when reasoning about safety because their effects should be invisible to other transactions.

[2]In the context of a relational database, global variables correspond to fields/rows of a table while in the context of a key-value store, they correspond to keys.

variable and storing it into a local variable a ($a := \texttt{read}(x)$) , writing the value of a local variable a to a global variable x ($\texttt{write}(x, a)$), or an assignment to a local variable a ($a := e$). The set of values of global or local variables is denoted by Vals. Assignments to local variables use expressions e over local variables, which are interpreted as values and whose syntax is left unspecified. Each of these instructions can be guarded by a Boolean condition $\phi(\vec{a})$ over a set of local variables $\vec{a}$ (their syntax is not important). Our results assume bounded programs, and therefore, we omit other constructs like `while` loops.

2.2 Isolation Level Definition

We present the axiomatic framework introduced by **?** for defining isolation levels in key-value stores.[3] Isolation levels are defined as logical constraints, called *axioms*, over *histories*, which are an abstract representation of the interaction between a program and the database in a concrete execution.

2.2.1 Histories. Programs interact with a database by issuing transactions formed of `begin`, end, read and write instructions. The effect of executing one such instruction is represented using an *event* $\langle e, type \rangle$ where e is an *identifier* and *type* is a *type*. There are four types of events: begin, commit, read(x) for reading the global variable x, and write(x, v) for writing value v to global variable x. The set of events is denoted by $\mathcal{E}$. For a read/write event e, we use $var(e)$ to denote the variable x.

A *transaction log* $\langle t, E, \mathsf{po}_t \rangle$ is an identifier t and a finite set of events E along with a strict total order po_t on E, called *program order*. The minimal element of po_t is a begin event. A transaction log without an commit event is called *pending*. Otherwise, it is called *complete*. If the commit event occurs, then it is maximal in po_t. The set E of events in a transaction log t is denoted by events(t).

The program order po_t represents the order between instructions in the body of a transaction. We assume that each transaction log is well-formed in the sense that if a read of a global variable x is preceded by a write to x in po_t, then it should return the value written by the last write to x before the read (w.r.t. po_t). This property is implicit in the definition of every isolation level that we are aware of. For simplicity, we may use the term *transaction* instead of transaction log.

The set of read(x) events in a transaction log t that are *not* preceded by a write to x in po_t, for some x, is denoted by reads(t). As mentioned above, the other read events take their values from writes in the same transaction and their behavior is independent of other transactions. Also, the set of write$(x, _)$ events in t that are *not* followed by other writes to x in po_t, for some x, is denoted by writes(t). If a transaction contains multiple writes to the same variable, then only the last one (w.r.t. po_t) can be visible to other transactions (w.r.t. any isolation level that we are aware of). The extension to sets of transaction logs is defined as usual. Also, we say that a transaction log t *writes* x, denoted by t writes x, when writes(t) contains some write$(x, _)$ event.

A *history* contains a set of transaction logs (with distinct identifiers) ordered by a (partial) *session order* so that represents the order between transactions in the same session.[4] It also includes a *write-read* relation (also called read-from) that "justifies" read values by associating each read to a transaction that wrote the value returned by the read.

[3]Isolation levels are called consistency models by **?**.
[4]In the context of our programming language, so would be a union of total orders. This constraint is not important for defining isolation levels.

Definition 2.1. A *history* $\langle T, \mathsf{so}, \mathsf{wr}\rangle$ is a set of transaction logs T along with a strict partial *session order* so, and a *write-read* relation $\mathsf{wr} \subseteq T \times \mathsf{reads}(T)$ such that

- the inverse of wr is a total function,
- if $(t, e) \in \mathsf{wr}$, where e is a $\mathsf{read}(x)$ event, then $\mathsf{writes}(t)$ contains some $\mathsf{write}(x, \_)$ event, and
- $\mathsf{so} \cup \mathsf{wr}$ is acyclic.

We assume that every history includes a distinguished transaction log writing the initial values of all global variables. This transaction log precedes all the other transaction logs in so. We use h, h_1, h_2, ... to range over histories.

For a variable x, wr_x denotes the restriction of wr to reads of x, i.e., , $\mathsf{wr}_x = \mathsf{wr} \cap (T \times \{e \mid e \text{ is a } \mathsf{read}(x) \text{ event}\})$. Moreover, we extend the relations wr and wr_x to pairs of transactions by $\langle t_1, t_2\rangle \in \mathsf{wr}$, resp., $\langle t_1, t_2\rangle \in \mathsf{wr}_x$, iff there exists a $\mathsf{read}(x)$ event e in t_2 such that $\langle t_1, e\rangle \in \mathsf{wr}$, resp., $\langle t_1, e\rangle \in \mathsf{wr}_x$. We say that the transaction log t_1 is *read* by the transaction log t_2 when $\langle t_1, t_2\rangle \in \mathsf{wr}$.

The set of transaction logs T in a history $h = \langle T, \mathsf{so}, \mathsf{wr}\rangle$ is denoted by $\mathsf{tr}(h)$, and the union of $\mathsf{events}(t)$ for $t \in T$ is denoted by $\mathsf{events}(h)$. Given a history h and an event e in h, $\mathsf{tr}(h, e)$ is the transaction t in h that contains e. Also, we define $\mathsf{writes}(h) = \bigcup_{t \in \mathsf{tr}(h)} \mathsf{writes}(t)$ and $\mathsf{reads}(h) = \bigcup_{t \in \mathsf{tr}(h)} \mathsf{reads}(t)$.

To simplify the notation, we extend so and wr to pairs of events by $(e_1, e_2) \in \mathsf{so}$ if $(\mathsf{tr}(h, e_1), \mathsf{tr}(h, e_2)) \in \mathsf{so}$ and $(e_1, e_2) \in \mathsf{wr}$ if $e_1 \in \mathsf{writes}(\mathsf{tr}(h, e_1))$, $(\mathsf{tr}(h, e_1), e_2) \in \mathsf{wr}$, and $var(e_1) = var(e_2)$. We also define $\mathsf{po} = \bigcup_{t \in T} \mathsf{po}_t$.

TODO NOT SURE THAT THE FOLLOWING DEFINITION IS NEEDED

Definition 2.2. 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 $e = \textsc{next}(h)$ is $\mathsf{begin}, \mathtt{write}$ or end, we will denote by $h \bullet e$ the history $h' = \langle E', \mathsf{so}', \mathsf{wr}\rangle$ where $E' = \mathsf{events}(h) \cup \{e\}$ and $\mathsf{so}' = \mathsf{so} \cup \{\langle e', e\rangle \mid e' \in h \wedge \mathtt{th}(e) = \mathtt{th}(e')\}$. On the other hand, if e is a $\mathtt{read}$ event, we will define the history $h'_w = h \bullet_w e$ for some $\mathtt{write}$ event $w \in h$ as $\langle E', \mathsf{so}', \mathsf{wr} \cup \{\langle w, r\rangle\}\rangle$; 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!!!!!!!

2.2.2 Axiomatic Framework. 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 2.3. 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.

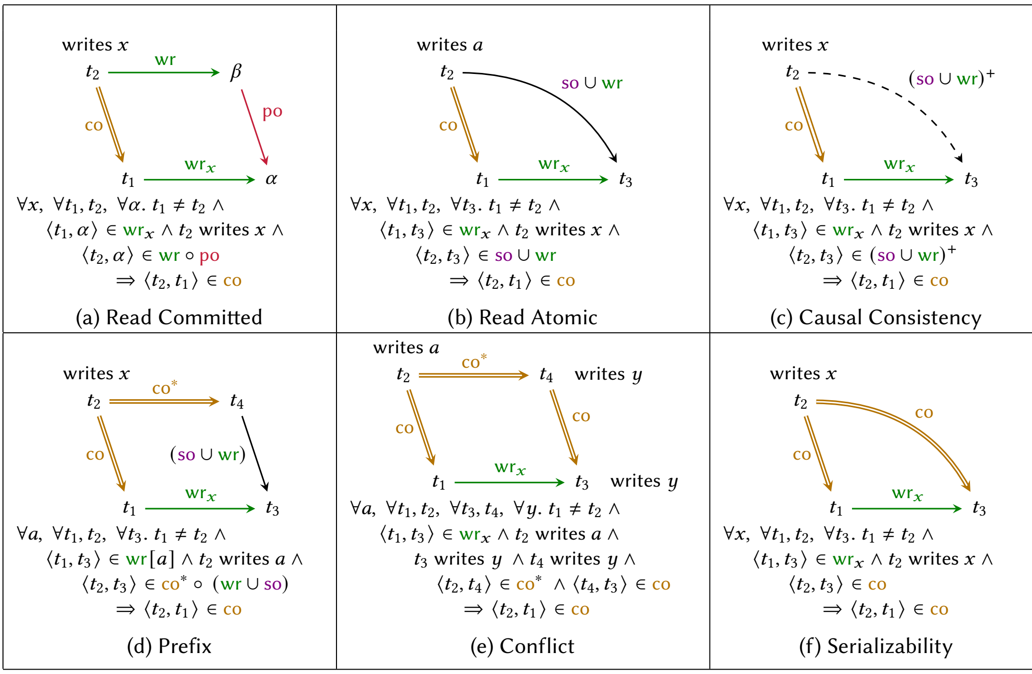


Fig. 2. Axioms defining isolations levels. The reflexive and transitive, resp., transitive, closure of a relation *rel* is denoted by rel^*, resp., rel^+. Also, $\circ$ denotes the composition of two relations, i.e., $rel_1 \circ rel_2 = \{\langle a, b\rangle | \exists c. \langle a, c\rangle \in rel_1 \wedge \langle c, b\rangle \in rel_2\}$.

In figure 2 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.

TODO DEFINE I-consistent histories

2.3 Program Semantics

We define a small-step operational semantics for transactional programs, which is parametrized by an isolation level I. The semantics keeps a history of previously executed database accesses in order to maintain conformance to I.

For readability, we define a program as a partial function $\mathsf{P} : \mathsf{SessId} \rightharpoonup \mathsf{Sess}$ that associates session identifiers in SessId with concrete code as defined in Figure 1 (i.e., sequences of transactions). Similarly, the session order so in a history is defined as a partial function $\mathsf{so} : \mathsf{SessId} \rightharpoonup \mathsf{Tlogs}^*$ that associates session identifiers with sequences of transaction logs. Two transaction logs are ordered by so if one occurs before the other in some sequence $\mathsf{so}(j)$ with $j \in \mathsf{SessId}$.

Formally, the operational semantics is defined as a transition relation $\Rightarrow_I$ between *configurations*, which are defined as tuples containing the following:

SPAWN

$$\frac{t \text{ fresh} \quad e \text{ fresh} \quad \mathsf{P}(j) = \mathsf{begin}; \mathsf{Body}; \mathsf{commit}; \mathsf{S} \quad \vec{\mathsf{B}}(j) = \epsilon}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h \oplus_j \langle t, \{\langle e, \mathsf{begin}\rangle\}, \emptyset\rangle, \vec{\gamma}[j \mapsto \emptyset], \vec{\mathsf{B}}[j \mapsto \mathsf{Body}; \mathsf{commit}], \mathsf{P}[j \mapsto \mathsf{S}]}$$

IF-TRUE

$$\frac{\psi(\vec{x})[x \mapsto \vec{\gamma}(j)(x) : x \in \vec{x}] \text{ true} \qquad \vec{\mathsf{B}}(j) = \mathtt{if}(\psi(\vec{x}))\{\mathsf{Instr}\}; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{Instr}; \mathsf{B}], \mathsf{P}}$$

IF-FALSE

$$\frac{\psi(\vec{x})[x \mapsto \vec{\gamma}(j)(x) : x \in \vec{x}] \text{ false} \qquad \vec{\mathsf{B}}(j) = \mathtt{if}(\psi(\vec{x}))\{\mathsf{Instr}\}; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

LOCAL

$$\frac{v = \vec{\gamma}(j)(e) \qquad \vec{\mathsf{B}}(j) = a := e; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h, \vec{\gamma}[(j, a) \mapsto v], \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

WRITE

$$\frac{v = \vec{\gamma}(j)(x) \quad e \text{ fresh} \quad \vec{\mathsf{B}}(j) = \mathtt{write}(x, a); \mathsf{B} \quad h \oplus_j \langle e, \mathsf{write}(x, v)\rangle \text{ satisfies } I}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h \oplus_j \langle e, \mathsf{write}(x, v)\rangle, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

READ-LOCAL

$$\frac{\mathsf{writes}(\mathit{last}(h, j)) \text{ contains a } \mathsf{write}(x, v) \text{ event} \qquad e \text{ fresh} \qquad \vec{\mathsf{B}}(j) = a := \mathsf{read}(x); \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h \oplus_j \langle e, \mathsf{read}(x)\rangle, \vec{\gamma}[(j, a) \mapsto v], \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

READ-EXTERN

$$\frac{\begin{array}{c}\mathsf{writes}(\mathit{last}(h, j)) \text{ does not contain a } \mathsf{write}(x, v) \text{ event} \qquad e \text{ fresh} \qquad \vec{\mathsf{B}}(j) = a := \mathsf{read}(x); \mathsf{B} \\ h = (T, \mathsf{so}, \mathsf{wr}) \qquad t = \mathit{last}(h, j) \qquad \mathsf{write}(x, v) \in \mathsf{writes}(t') \text{ with } t' \in \mathsf{compTrans}(h) \text{ and } t \neq t' \\ h' = (h \oplus_j \langle e, \mathsf{read}(x)\rangle) \oplus \mathsf{wr}(t', e) \qquad h' \text{ satisfies } I\end{array}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h', \vec{\gamma}[(j, a) \mapsto v], \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

COMMIT

$$\frac{e \text{ fresh} \quad \vec{\mathsf{B}}(j) = \mathtt{commit}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h \oplus_j \langle e, \mathtt{commit}\rangle, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \epsilon], \mathsf{P}}$$

Fig. 3. An operational semantics for transactional programs. Above, $\mathit{last}(h, j)$ denotes the last transaction log in the session order $\mathsf{so}(j)$ of h, and $\mathsf{compTrans}(h)$ denotes the set of transaction logs in h that are complete.

- history h storing the events generated by database accesses executed in the past,
- a valuation map $\vec{\gamma}$ that records local variable values in the current transaction of each session ($\vec{\gamma}$ associates identifiers of sessions that have live transactions with valuations of local variables),
- a map $\vec{B}$ that stores the code of each live transaction (associating session identifiers with code), and
- sessions/transactions P that remain to be executed from the original program.

Before presenting the definition of $\Rightarrow_I$, we introduce some notation. Let h be a history that contains a representation of so as above. We use $h \oplus_j \langle t, E, \mathsf{po}\rangle$ to denote a history where $\langle t, E, \mathsf{po}\rangle$ is appended to $\mathsf{so}(j)$. Also, for a event e, $h \oplus_j e$ is the history obtained from h by adding e to the last transaction log in $\mathsf{so}(j)$ and as a last event in the program order of this log (i.e., if $\mathsf{so}(j) = \sigma; \langle t, E, \mathsf{po}\rangle$, then the session order so' of $h \oplus_j e$ is defined by $\mathsf{so}'(k) = \mathsf{so}(k)$ for all $k \neq j$ and $\mathsf{so}(j) = \sigma; \langle t, E \cup \{e\}, \mathsf{po} \cup \{(e', e) : e' \in E\}\rangle$). Finally, for a history $h = \langle T, \mathsf{so}, \mathsf{wr}\rangle$, $h \oplus \mathsf{wr}(t, e)$ is the history obtained from h by adding (t, e) to the write-read relation.

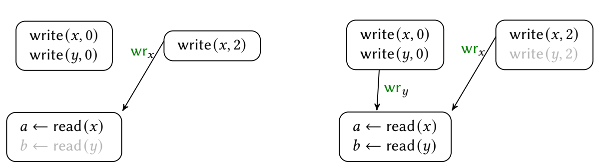
Figure 3 lists the rules defining $\Rightarrow_I$. SPAWN starts a new transaction in a session j provided that this session has no live transaction ($\vec{\mathsf{B}}(j) = \epsilon$). It adds a transaction log with a single begin event to the history and schedules the body of the transaction. IF-TRUE and IF-FALSE check the truth value of a Boolean condition of an `if` conditional. LOCAL models the execution of an assignment to a local variable which does not impact the stored history. READ-LOCAL and READ-EXTERN concern read instructions. READ-LOCAL handles the case where the read follows a write on the variable x in the same transaction: the read returns the value written by the last write on x in that transaction. Otherwise, READ-EXTERN corresponds to reading a value written in another transaction t'. The transaction t' is chosen non-deterministically as long as extending the current history with the write-read dependency associated to this choice leads to a history that still satisfies I. READ-EXTERN applies only when the executing transaction contains no write on the same variable.

An *initial* configuration for program P contains the program P along with a history $h = \langle \{t_0\}, \emptyset, \emptyset\rangle$, where t_0 is a transaction log containing only writes that write the initial values of all variables, and empty current transaction code ($\mathsf{B} = \epsilon$). An execution of a program P under an isolation level I is a sequence of configurations $c_0 c_1 \ldots c_n$ where c_0 is an initial configuration for P, and $c_m \Rightarrow_I c_{m+1}$, for every $0 \leq m < n$. We say that c_n is *I-reachable* from c_0. The history of such an execution is the history h in the last configuration c_n. A configuration is called *final* if it contains the empty program ($\mathsf{P} = \emptyset$). Let $\mathsf{hist}_I(\mathsf{P})$ denote the set of all histories of an execution of P under I that ends in a final configuration.

3 PREFIX-CLOSED AND CAUSALLY-EXTENSIBLE ISOLATION LEVELS

TODO THE INTUITION BEHIND THIS CONDITION (USE THE TEXT THAT FOLLOWS)

Besides models presented in figure 2, 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. 4. Example of a dead-lock after swapping two events.

Let's analyze the histories h_1 and h_2 described in figure 4a and 4b 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 $[\mathsf{wr}]$ 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 $\mathsf{so} \cup \mathsf{wr}$ 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, $t(r)$ is $\mathsf{so} \cup \mathsf{wr}$-maximal while in h_2 $t(w)$ is not.

As real database executions forbid transactions reading from non-committed ones, it is reasonable to allow those transactions $\mathsf{so} \cup \mathsf{wr}$-maximal to be executed completed without hindering the previous committed transactions.

For a relation $R \subseteq A \times A$, the restriction of R to $A' \times A'$, denoted by $R \downarrow A' \times A'$, is defined by $\{(a, b) : (a, b) \in R, a, b \in A'\}$. Also, a set A' is called R-downward closed when it contains $a \in A$ every time it contains some $b \in A$ with $(a, b) \in R$.

A *prefix* of a transaction log $\langle t, E, \mathsf{po}_t \rangle$ is a transaction log $\langle t, E', \mathsf{po}_t \downarrow E' \times E' \rangle$ such that E' is po_t-downward closed. A *prefix* of a history $h = \langle T, \mathsf{so}, \mathsf{wr} \rangle$ is a history $h' = \langle T', \mathsf{so} \downarrow T' \times T', \mathsf{wr} \downarrow T' \times T' \rangle$ such that every transaction log in T' is a prefix of a different transaction log in T but carrying the same id, and $\mathsf{events}(h') \subseteq \mathsf{events}(h)$ is $(\mathsf{po} \cup \mathsf{so} \cup \mathsf{wr})^*$-downward closed. TODO GIVE AN EXAMPLE OF A PREFIX ON THE FIGURE ABOVE.

Definition 3.1. An isolation level I is called *prefix-closed* when every prefix of an I-consistent history is also I-consistent.

THEOREM 3.2. *Every model depicted in figure 2 is prefix closed.*

PROOF. Let h be a consistent history. As any $\mathsf{so} \cup \mathsf{wr}$-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'. □

Let $h = \langle T, \mathsf{so}, \mathsf{wr} \rangle$ be a history. A transaction t is called $(\mathsf{so} \cup \mathsf{wr})^*$-maximal in h if h does not contain any transaction t' such that $(t, t') \in (\mathsf{so} \cup \mathsf{wr})^*$. We define a *causal extension* of a pending transaction t in h with an event e as follows:

- e is added to t as a maximal element of po_t,
- if e is a read event, then wr is extended with some tuple (t', e) such that $t'\ [\mathsf{so} \cup \mathsf{wr}]^*\ t$
- the other elements of h remain unchanged.

TODO NOTE THAT A HISTORY MAY HAVE MULTIPLE CAUSAL EXTENSIONS WITH A READ. NOT FOR THE OTHER TYPES OF EVENTS. GIVE EXAMPLES.

Definition 3.3. An isolation level I is called *causally-extensible* if for every I-consistent history h, every $(\mathsf{so} \cup \mathsf{wr})^*$-maximal pending transaction t in h, and every event e, there exists a causal extension of t with e that is I-consistent.

THEOREM 3.4. *Causal Consistency* (CC)*, Read Atomic* (RA) *and Read Committed* (RC) *are causally-extensible.*

PROOF. Let $\mathcal{I}$ an isolation level in $\{\mathsf{CC}, \mathsf{RA}, \mathsf{RC}\}$, h a non-total consistent history and let e a $\mathsf{so} \cup \mathsf{wr}$-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 $\mathsf{co}' = \mathsf{co} \cup \{\langle T, t(e) \rangle, T \in h\}$ 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 $\mathsf{so} \cup \mathsf{wr}$-maximal `read` event that reads variable x and let us find a `write` event w s.t. $t(w)\ [\mathsf{so} \cup \mathsf{wr}]^*\ t(r)$ 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. $t(w)$ $[\mathsf{so} \cup \mathsf{wr}]^*$ $t(r)$; 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 $t(w_{n+1})$ $[\mathsf{so} \cup \mathsf{wr}]^*$ $t(r)$, if there were a cycle, it would be due to a transaction T such that writes x, $t(w_{n+1})$ $[\mathsf{co}]^*$ T and $\varphi_{\mathcal{M}}(T, e)$, where $\varphi_{\mathsf{CC}}(T, e) = T$ $[\mathsf{so} \cup \mathsf{wr}]^+$ $t(e)$, $\varphi_{\mathsf{RA}}(T, e) = T$ $[\mathsf{so} \cup \mathsf{wr}]$ $t(e)$ and $\varphi_{\mathsf{RC}}(T, e) = T$ $[\mathsf{wr} \circ \mathsf{po}]$ e. In particular, for any of the three models, T $[\mathsf{so} \cup \mathsf{wr}]^*$ $t(r)$. 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 $[\mathsf{wr}_x]$ $t(r)$, the cycle that was between T and $t(w_{n+1})$ 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 $[\mathsf{co}]$ T' and $\varphi_{\mathcal{M}}(T', r)$. Therefore, in $h_{w_{n+1}}$ $t(w_{n+1})$ $[\mathsf{co}]$ T $[\mathsf{co}]$ T', so there is a cycle between in $t(w_{n+1})$ 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** $[\mathsf{so} \cup \mathsf{wr}]^*$ $t(r)$ 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 3.5. *Prefix Consistency (*PRE*) is maximally-extensible.*

Proof. Let h a non-total PRE-consistent history, co a commit order that witness this property and e a $\mathsf{so} \cup \mathsf{wr}$-maximal event. As if e is the begin of a transaction, $\mathsf{co}' = \mathsf{co} \cup \{\langle t, \mathsf{tr}(h, e)\rangle, t \in h\}$ is a witness of $h \oplus e$'s PRE-consistency; let us show that if $\mathtt{begin}(\mathsf{tr}(h, e)) \in h$, there is a commit order co' for h. If that would be the case, $h \oplus e$ will be consistent with co' as witness.

Let $\mathsf{co}' = \{\langle t, t'\rangle \mid t \times t' \in h^2 \text{ s.t. } t\ [\mathsf{co}]\ t' \land t \neq \mathsf{tr}(h, e)\} \cup \{\langle t, \mathsf{tr}(h, e) \mid t \in h\rangle\}$. As co' is a total order between transactions, co' witness h is PRE-consistent if and only if there is no t_1, t_2, t_3, t_4 transactions such that $(t_1, t_3) \in \mathsf{wr}_x$, t_2 writes x, $(t_2, t_4) \in \mathsf{co}'^*$ and $(t_4, t_3) \in \mathsf{so} \cup \mathsf{wr}$ but $(t_1, t_2) \in \mathsf{co}'$. If $\mathsf{tr}(h, e)$ is not either t_1, t_2, t_3 or t_4, as co' only modifies the relative order between $\mathsf{tr}(h, e)$ and every other transaction, this situation cannot happen. On one hand, as $\mathsf{tr}(h, e)$ is $\mathsf{so} \cup \mathsf{wr}$-maximal, $\mathsf{tr}(h, e)$ cannot be t_1 nor t_4. On the other hand, as $\mathsf{tr}(h, e)$ is co'-maximum, it cannot be also t_1. Finally, if $t_3 = \mathsf{tr}(h, e)$, then the exact situation would happen regarding co, which is impossible. Therefore, co' witness h is PRE-consistent. □

TODO SHOW THAT SERIALIZABILITY IS NOT CAUSALLY-EXTENSIBLE.

4 SWAPPING-BASED MODEL CHECKING ALGORITHMS

We present a class of model checking algorithms for enumerating executions of a given transactional program, that we call *swapping-based algorithms*. Section 5 will describe a concrete instance that applies to isolation levels that are prefix-closed and causally extensible.

These algorithms are defined by the recursive function EXPLORE listed in Algorithm 1. The function EXPLORE receives as input a program P, an *ordered history* $h_<$, which is a pair $(h, <)$ of a history and a total order $<$ on all the events in h, and a mapping locals that associates each event e in h with the valuation of local variables in the transaction of e ($\mathsf{tr}(h, e)$) just before executing e. For an ordered history $(h, <)$ with $h = \langle T, \mathsf{so}, \mathsf{wr}\rangle$, we assume that $<$ is consistent with po, so, and wr,

Algorithm 1 EXPLORE algorithm

function EXPLORE(P, $h_<$, locals)
  $j, e, \gamma \leftarrow$ NEXT(P, $h_<$, locals)
  locals′ $\leftarrow$ locals$[e \mapsto \gamma]$
  if $e = \bot$ and VALID(h) **then**
    output h, locals′
  else if $type(e) =$ `read` **then**
    for all $w \in$ VALIDWRITES(h, e) **do**
      $h'_< \leftarrow h_< \oplus_j e \oplus \mathsf{wr}(w, e)$
      EXPLORE(P, $h'_<$, locals′)
      EXPLORESWAPS(P, $h'_<$, locals′)
  else
    $h'_< \leftarrow h_< \oplus_j e$
    EXPLORE(P, $h'_<$, locals′)
    EXPLORESWAPS(P, $h'_<$, locals′)
  end if
end function
function EXPLORESWAPS(P, $h_<$, locals)
  $l \leftarrow$ COMPUTEREORDERINGS($h_<$)
  for all $(\alpha, \beta) \in l$ **do**
    if OPTIMALITY($h_<, \alpha, \beta$) **then**
      EXPLORE(P, SWAP($h_<, \alpha, \beta$, locals))
    end if
end function

i.e., $e_1 < e_2$ if $(\mathsf{tr}(h, e_1), \mathsf{tr}(h, e_2)) \in (\mathsf{so} \cup \mathsf{wr})^+$ or $(e_1, e_2) \in \mathsf{po}$. Initially, the ordered history and the mapping locals are empty.

The function EXPLORE calls NEXT to obtain an event representing the next database access in some pending transaction of the input program. A typical implementation of NEXT would choose one of the pending transactions, execute all local instructions until the next database instruction in that transaction (applying the transition rules IF-TRUE, IF-FALSE, and LOCAL from Figure 3) and return the event e corresponding to that database instruction and the current local state γ. NEXT may also return $\bot$ if the program finished. If NEXT returns $\bot$, then the function VALID can be used to filter executions that satisfy the intended isolation level before outputting the current history and local states.

Otherwise, the event e is added to the ordered history $h_<$. If e is a read event, then VALIDWRITES computes a set of write events w in the current history that are valid for e, i.e., adding the event e along with the wr dependency (w, e) leads to a history that still satisfies the intended isolation level. Concerning notations, we extend $\oplus_j$ to ordered histories in a straightforward manner: $(h, <) \oplus_j e$ is the ordered history $(h \oplus_j e, < \cdot e)$ where $< \cdot e$ means that e is added as the maximal element of the total order $<$. Also, $h \oplus_j (e, \text{begin})$ is the same as $h \oplus_j \langle t, \{\langle e, \text{begin}\rangle\}, \emptyset\rangle$ for a fresh transaction id t. Moreover, we simplify the notation and write $h \oplus \mathsf{wr}(w, e)$ for w a write event and e a read event instead of $h \oplus \mathsf{wr}(\mathsf{tr}(h, w), e)$.

Moreover, once an event is added to the current history, the algorithm may explore other histories obtained by re-ordering events in the current one. Such re-orderings are required for completeness.

As explained above, new read events can only read from writes executed in the past which limits the set of explored histories to the scheduling imposed by NEXT. Without re-orderings, write events scheduled later by NEXT cannot be read by read events executed in the past, even-though this may be permitted by the intended isolation level.

The function EXPLORESWAPS calls COMPUTEREORDERINGS to compute pairs of sequences of events α, β that should be re-ordered; α and β are *contiguous and disjoint* subsequences of the total order $<$, and α should end before β (since β will be re-ordered before α). Typically, α should contain a read event r and β a write event w such that re-ordering the two enables r to read from w. Avoiding redundancy, i.e., exploring the same history multiple times, may require restricting the application of such re-orderings. This is modeled by the Boolean condition called OPTIMALITY. If this condition holds, the new explored histories are computed by the function SWAP. This function returns local states as well, which are necessary for continuing the exploration. We assume that $\textsc{Swap}((h, <), \alpha, \beta, \mathsf{locals})$ returns pairs $((h', <'), \mathsf{locals}')$ such that

- h' contains the events in α and β,
- h' without the events in α is a prefix of h, and
- for every read event r in α, if it reads from different writes in h and h' (i.e., the writes w_1 and w_2 associated with r by the wr relations of h and h', respectively, are different) then it is the last event in its transaction log (w.r.t. po).

The first condition makes the re-ordering "meaningful", while the last two conditions ensure that the history h' is feasible (i.e., it can be obtained using the operational semantics defined in Section 2.3) and restricts the possible changes of the wr relation to events in α. Concerning the events in β, they imply that h' contains all their $(\mathsf{po} \cup \mathsf{so} \cup \mathsf{wr})^*$ predecessors. The last condition is required since changing the value returned by a read access may disable later events in the same transaction.

A concrete implementation of EXPLORE is called:

- *I-sound* if it outputs only histories in $\mathsf{hist}_I(\mathsf{P})$ for every program P,
- *I-complete* if it outputs every history in $\mathsf{hist}_I(\mathsf{P})$ for every program P,
- *optimal* if it does not output the same history twice,
- *strongly optimal* if it is optimal and never engages in fruitless explorations, i.e., EXPLORE is never called (recursively) on a history h that does not satisfy I, and every call to EXPLORE results in an output or a recursive call to EXPLORE.

5 SWAPPING-BASED MODEL CHECKING FOR PREFIX-CLOSED AND CAUSALLY-EXTENSIBLE ISOLATION LEVELS

We present a concrete implementation of EXPLORE that is I-sound, I-complete, and strongly optimal for any isolation level I that is prefix-closed and causally-extensible. Moreover, the space complexity of this algorithm is polynomial in the size of the program. An important invariant of this implementation is that it explores histories with *at most one* pending transaction and this transaction is maximal w.r.t. session order. This invariant is used to avoid fruitless explorations: since I is assumed to be causally-extensible, there always exists an extension of the current history with one more event that continues to satisfy I. We prove the correctness of the algorithm in Section 8.

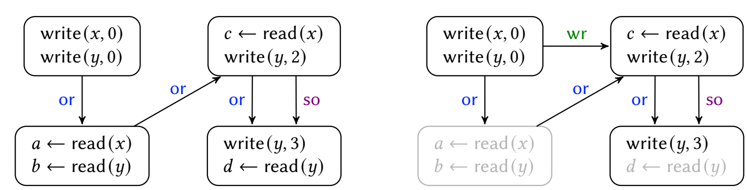
5.1 Extending Histories According to An Oracle Order

The function NEXT generating events that represent database accesses is parametrized by an *arbitrary but fixed* order between the transactions in the program called *oracle order*. This order, denoted as $<_{\mathsf{or}}$ and trivially extensible to events in a history, has to respect the order between transactions in the same session of the program.

NEXT returns a new event of the transaction that is not already completed and that is *minimal* according to $<_{\mathsf{or}}$. In more detail, if j, e, γ is the output of $\textsc{Next}(\mathsf{P}, h_<, \mathsf{locals})$, then either:

- the last transaction log t of session j (w.r.t. so) in h is pending, and t is the smallest among pending transaction logs in h w.r.t. $<_{\mathsf{or}}$
- h contains no pending transaction logs and the next transaction of sessions j is the smallest among not yet started transactions in the program.

This implementation of NEXT is deterministic and it prioritizes the completion of pending transactions. The latter is useful to maintain the invariant that any history explored by the algorithm has at most one pending transaction. Preserving this invariant requires that the histories given as input to NEXT also have at most one pending transaction. This is discussed further when explaining the process of re-ordering events in Section 5.2.



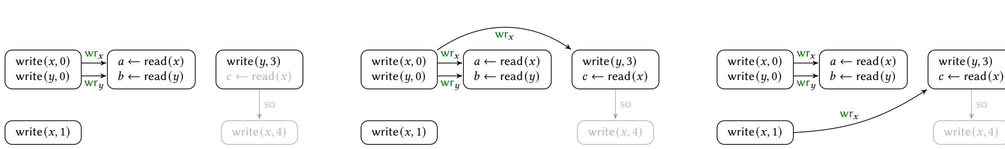
(a) or produce a total order between all transactions in the program.

(b) An incomplete history and its or order.

Fig. 5. Some possible oracle order between transactions. TODO MAY NEED TO EXPLAIN THE CONVENTIONS IN THE PICTURE.

TODO: THIS SHOULD BE AN EXAMPLE OF A STEP OF NEXT. ADD TO THE ABOVE EXAMPLE SOME LOCAL INSTRUCTION BEFORE THE READ(Y), AND TALK ABOUT LOCAL STATES AS WELL. ALSO, ON THE LEFT, PUT THE PROGRAM AND NOT A HISTORY (PROGRAM WRITTEN WITH THE SYNTAX PRESENTED AT THE BEGINNING OF THE PAPER).

For example, given the history h in Figure 5b, the function NEXT would return the event $d \leftarrow \mathsf{read}(y)$ instead of $a \leftarrow \mathsf{read}(x)$; as the forth transaction is pending in h.



(a) Current history.

(b) One possible extension.

(c) Another possible extension.

Fig. 6. Extensions of a history by adding a read event. TODO MAY NEED TO EXPLAIN THE CONVENTIONS IN THE PICTURE.

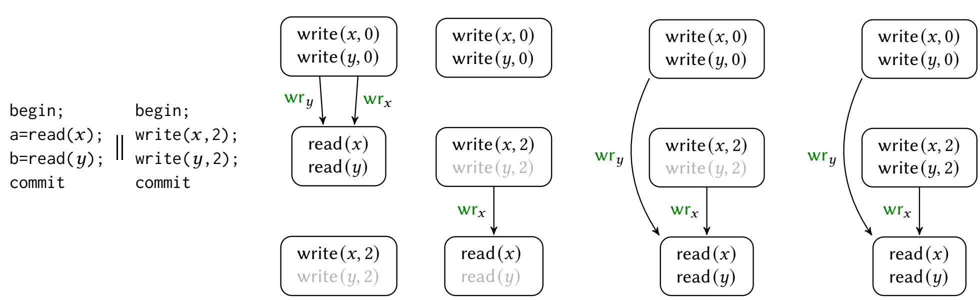
TODO REDO THE ABOVE FIGURE IN THE SPIRIT OF FIGURE 7. MODIFY THE EXAMPLE SO ONE OF THE TWO CHOICES DOES NOT SATISFY SOME ISOLATION LEVEL, TO EXEMPLIFY THE USE OF VALIDWRITES.

If the event returned by NEXT is not a read event, then it is simply added to the current history, as the maximal element of the order $<$ (cf. the definition of $\oplus_j$ on ordered histories). If it is a read event, then adding this event may result in multiple histories depending on the associated wr dependency. For example, in Figure 6, extending the history in Figure 6a with the read(x) event could result in two different histories, pictured in Figure 6b and 6c, depending on the write with whom this read event is associated by wr. The function VALIDWRITES limits the choices to those that preserve consistency with the intended isolation level I, i.e.,

$$\textsc{ValidWrites}(h, e) \quad := \quad \{w \mid h \oplus_j e \oplus \mathsf{wr}(w, e) \text{ satisfies } I\}$$

5.2 Re-Ordering Events in Histories

After extending the current history with one more event, EXPLORE may recurse on other histories obtained by re-ordering events in the current one (and dropping some other events).



(a) Program. (b) Current. (c) Swapped. (d) Extended. (e) Inconsistent.

Fig. 7. Example of inconsistency after swapping two events. TODO MAY NEED TO EXPLAIN THE CONVENTIONS IN THE PICTURE.

Re-ordering events must preserve the invariant of producing histories with at most one pending transaction. To explain the use of this invariant in avoiding fruitless explorations, let us consider the program in Figure 7a assuming an exploration under Read Committed. The oracle order gives priority to the transaction on the left. Assume that the current history reached by the exploration is the one pictured in Figure 7b (the last added event is write$(x, 2)$). Swapping the write$(x, 2)$ event with the read(x) event would result in the history pictured in Figure 7c. To ensure that this swap produces a new history which was not explored in the past, the wr_x dependency of read(x) is changed towards the write$(x, 2)$ transaction (we detail this later). By the definition of NEXT (and the oracle order), this history shall be extended with read(y), and this read event will be associated by wr_y to the only available `write`$(y, \_)$ event. This is pictured in Figure 7d. The next exploration step will extend the history with write$(x, 2)$ (the only extension possible) which however, results in a history that does *not* satisfy Read Committed, thereby, the recursive exploration branch being blocked. The core issue is related to the history in Figure 7d which has a pending transaction that is *not* $(\mathsf{so} \cup \mathsf{wr})^*$-maximal. Being able to extend such a transaction while maintaining consistency with the intended isolation level is not guaranteed by Read Committed (and any other isolation level we consider). Nevertheless, causal extensibility guarantees the existence of an extension for pending transactions that are $(\mathsf{so} \cup \mathsf{wr})^*$-maximal. We enforce this requirement by restricting the explored histories to have at most one pending transaction. This pending transaction will necessarily by $(\mathsf{so} \cup \mathsf{wr})^*$-maximal.

To enforce histories with at most one pending transaction, the function ComputeReorderings, which identifies events to reorder, has a non-empty return value only when the last added event is commit (the end of a transaction). Therefore, in such a case, it returns pairs of read and write events on the same variable, the write event coming from the last completed transaction, and such that the transactions containing the two events are not causally dependent (i.e., related by $[\mathsf{so} \cup \mathsf{wr}]^*$).

$$
\begin{aligned}
\textsc{ComputeReorderings}(h_<) \quad := \quad & \{(r, w) \in \mathcal{E} \mid r < w \land r \in \mathsf{reads}(t) \land w \in \mathsf{writes}(t) \\
& \qquad \land \mathsf{var}(r) = \mathsf{var}(w) \\
& \qquad \land (\mathsf{tr}(h, r), t) \notin [\mathsf{so} \cup \mathsf{wr}]^* \land t \text{ is complete}\} \\
& \text{where } t \text{ is the transaction log of the last event in } <
\end{aligned}
$$

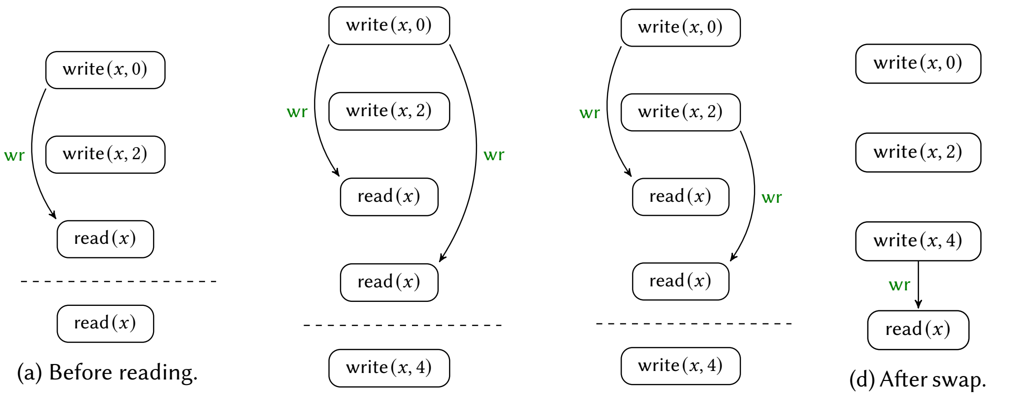
TODO GIVE AN EXAMPLE: A HISTORY, EMPHASIZING THE LAST COMPLETED TRANSACTION, AND THE OUTPUT OF GENERIC COMPUTE. MAKE IT SO THE OUTPUT CONTAINS TWO PAIRS. AND IT CAN BE USED TO EXEMPLIFY SWAP AS WELL (SOME THINGS TO KEEP AS DEPENDENCIES, SOME THINGS TO DELETE, INCLUDING EVENTS THAT FOLLOW THE READ IN THE SAME TRANSACTION.

Given a pair (r, w), the function Swap produces a new history h' which contains all events ordered before r (w.r.t. $<$), the transaction that contains w and all its $[\mathsf{so} \cup \mathsf{wr}]^*$ predecessors, and the event r reading from w. Note that the po predecessors of r from the same transaction are ordered before r by $<$ and they will be also included in h'. More generally, the history h' without r is a prefix of the input history h. By definition, the only pending transaction in h' is the one containing the read r. The order relation is updated by moving the transaction containing the read r to be the last; it remains unchanged for the rest of the events.

$$
\begin{aligned}
\textsc{Swap}(h_<, r, w, \mathsf{locals}) \quad := \quad & \big(h' = (h \setminus D) \oplus \mathsf{wr}(w, r), <' \big), \mathsf{locals}', \text{ where} \\
& D = \{e \mid r < e \land (\mathsf{tr}(h, e), \mathsf{tr}(h, w)) \notin [\mathsf{so} \cup \mathsf{wr}]^*\} \\
& <' = \big(< \downarrow (\mathsf{events}(h') \setminus \mathsf{events}(\mathsf{tr}(h', r)))\big) \cdot \mathsf{tr}(h', r) \\
& \mathsf{locals}' = \mathsf{locals} \downarrow \mathsf{events}(h').
\end{aligned}
$$

Above, $h \setminus D$ denotes the prefix of h obtained by deleting all the events in D from its transaction logs; a transaction log is removed all together if it becomes empty. Also, $h'' \oplus \mathsf{wr}(w, r)$ denotes an *update* of the wr relation of h'' where any pair $(\_, r)$ is replaced by (w, r). Finally, $<'' \cdot \mathsf{tr}(h', r)$ denotes an extension of the total order $<''$ obtained by appending the events in $\mathsf{tr}(h', r)$ according to program order.

TODO CONTINUE THE EXAMPLE ABOVE SHOWING THE RESULT OF SWAP IN ONE CASE.



(a) Before reading. (b) Reading 0. (c) Reading 2. (d) After swap.

Fig. 8. Re-ordering events versus optimality. Every history starts with a transaction writing the initial value 0.

5.3 Avoiding Redundancy

While extending histories according to Next and recursing on re-ordered histories whenever possible (taking Optimality as *true*) guarantees soundness and completeness, it does not guarantee optimality. There are two sources of redundancy in this trivial algorithm: (1) re-ordering the same read multiple times, and (2) applying Swap on different histories may give the same result.

TODO GIVE AN EXAMPLE FOR THE 1ST ISSUE. JUSTIFYING THE SWAPPED CONDITION.

Without further restrictions on swaps, it is also possible that different branches of the recursion lead to the same history, thereby, violating optimality. As an example, consider a program with 2 transactions that only read some variable x and 2 transactions that only write on x. Assume that explore reaches the ordered history pictured in Figure 8a and Next is about to return the second reading transaction. explore will recurse on the two histories pictured in Figure 8b and Figure 8c that differ in the write that this last read is reading from (either the initial write or the first write transaction). On both branches of the recursion, Next will extend the history with the last write transaction. For both histories, swapping this last write with the first read on x will result in the history pictured in Figure 8d (cf. the definition of ComputeReorderings and Swap). Therefore, both branches of the recursion will continue extending the same history and optimality is violated. The source of non-optimality is related to wr dependencies that are *removed* when computing the result of Swap. The histories in Figure 8b and Figure 8c were different because of the wr dependency involving the last read, but this difference was discarded during the computation of Swap. Therefore, we will restrict the application of Swap on histories where the discarded wr dependencies relate to some "fixed" set of writes, i.e., latest writes (w.r.t. $<$) that are valid, i.e., preserve consistency with the intended isolation level (see the definition of $\text{readsLatest}_I(\_,\_)$ below), e.g., when the second read reads from write$(x, 2)$.

The Optimality condition, which restricts re-orderings, requires that every read that will be deleted by Swap or the re-ordered read r (whose wr dependency will be modified) is not already swapped and it reads from a latest valid write (D has the same definition as in Swap):

$$\textsc{Optimality}(h_<, r, w) := \forall r' \in \mathsf{reads}(h) \cap (D \cup \{r\}).\ \neg\textsc{swapped}(h_<, r') \wedge \mathsf{readsLatest}_I(h_<, r')$$

We say that a read r is *swapped* in $h_<$ when (1) it reads from a write w that is a successor in the oracle order (the write was added by Next after the read), which is now a predecessor[5] in the history order $<$ (2) there is no transaction t that is before r in both the oracle order $<_{\mathsf{or}}$ and the history order $<$, and which is a $[\mathsf{so} \cup \mathsf{wr}]^+$ successor of w's transaction, and (3) r is the first read in its transaction to read from w. Formally,

$$\textsc{swapped}(h_<, r) \quad := \quad \begin{array}{c} w < r \wedge w >_{\mathsf{or}} r \\ \wedge \\ \forall e \in h.\ \mathsf{tr}(h, e) <_{\mathsf{or}} \mathsf{tr}(h, r) \Rightarrow (r < e \vee (\mathsf{tr}(h, w), \mathsf{tr}(h, e)) \notin [\mathsf{so} \cup \mathsf{wr}]^+) \\ \wedge \\ \forall r' \in \mathsf{reads}(h).\ (\mathsf{tr}(h, w), r') \in \mathsf{wr} \Rightarrow (r', r) \notin \mathsf{po} \end{array}$$
$$\text{where } w \text{ is the write event such that } (w, r) \in \mathsf{wr}$$

A read r reads from a latest valid write, denoted as $\mathsf{readsLatest}_I(h_<, r)$, if reading from any other later write w.r.t. $<$ violates the isolation level I. Formally,

$$\mathsf{readsLatest}_I(h_<, r) \quad := \quad \forall w' \in \mathsf{writes}(h).\ \ w < w' < r \Rightarrow (h \setminus \{e \mid r < e\}) \oplus \mathsf{wr}(w', r) \not\models I)$$
$$\text{where } w \text{ is the write event such that } (w, r) \in \mathsf{wr}$$

5.4 Outputting histories

As we show in Section 8, the algorithm described above is I-sound assuming that $\textsc{Valid}(h) ::= \mathit{true}$. Therefore, any history computed by applying the functions described above will satisfy the intended isolation level I.

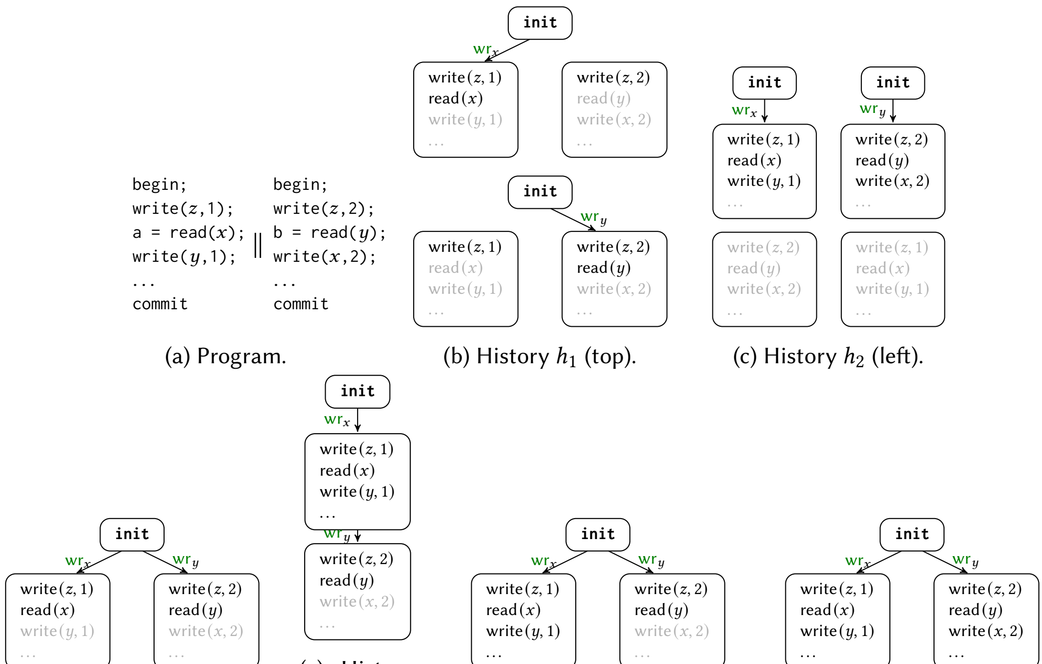
TODO THIS SHOULD PROBABLY BE ABOUT CORRECTNESS, STATING IT AND EXPLAINING A LITTLE BIT THE PROOFS.

6 WEAK DPOR ALGORITHMS FOR SNAPSHOT ISOLATION AND SERIALIZABILITY

As show in section 8, algorithm **??**'s completeness proof (theorem 8.15) 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 **??** behaves for the program depicted in figure **??** under them. We study this example as it has exactly two consistent executions but a dead-lock one under SI and SER.

Theorem 6.1. *If I is Snapshot Isolation or Serializability, there exists no* explore *algorithm that is I-sound, I-complete, and strongly optimal.*

[5] The explore maintains the invariant that every read follows the write it reads from in the history order $<$.



(a) Program. (b) History h_1 (top). (c) History h_2 (left).

(d) History h_{11}. (e) History $h_{12} = h_{21}$. (f) History $h'_{12} = h'_{21}$. (g) History h.

Fig. 9. A program and some partial histories. Events in grey are not yet added to the history. For h_{12} and h, the number of events that follow write(y, 1) and write(x, 2) is not important (we use black . . . to signify that).

Proof. We consider the program in Figure 9a, and show that any concrete instance of the explore function in Algorithm 1 *can not be both I*-complete and strongly optimal. This program contains two transactions, where only the first 3 instructions in each transaction are important. We show that if explore is I-complete, then it will necessarily be called recursively on a history h like in Figure 9g which does not satisfy I, thereby violating strong optimality. In the history h, both *Snapshot Isolation* and *Serializability* forbid the two reads reading initial values while the writes following them are also executed (committed).

Assuming that the function Next is not itself blocking (which would violate strong optimality), the explore will be called recursively on *exactly one* of the four histories in Figure 9b and Figure 9c, depending on which of the two reads is returned first by Next and the order defined by Next between the writes. We will continue our discussion with the history h_1 on the top of Figure 9b and the history h_2 on the left of Figure 9c. The other cases are similar (symmetric).

From h_1, explore can be called recursively either on h_{11} in Figure 9d, or on h_{12} and h'_{12} in Figure 9e and Figure 9f, depending on the order defined by Next between read(y) and write(y, 1) (read(y) is returned by Next before write(y, 1) in h_{11} and vice-versa in h_{12} and h_{22}). The histories h_{12} and h'_{12} differ in the read-from associated to read(y), and exploring at least h'_{12} is the best scenario towards ensuring I-completeness. If explore is called recursively only on h_{12}, then I-completeness is violated because h_{12} and any extension does not enable any re-ordering, and the history where read(x) reads from write(x, 2) will never be explored. Indeed, the two transactions in h_{12} are related by wr and events can be re-ordered earlier only together with their $(\mathsf{so} \cup \mathsf{wr})^*$ predecessors. From histories h_{11} and h'_{12}, EXPLORE will necessarily be called recursively on a history h like in Figure 9g which does not satisfy I, thereby violating strong optimality.

From h_2, EXPLORE can be called recursively on h_{21} in Figure 9e and h'_{21} in Figure 9f. As explained above for $h_{12} = h_{21}$, being called recursively only on h_{21} violates I-completeness, while being called recursively on $h'_{21} = h'_{12}$ leads to an inconsistent history, thereby violating strong optimality. □

Present code

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

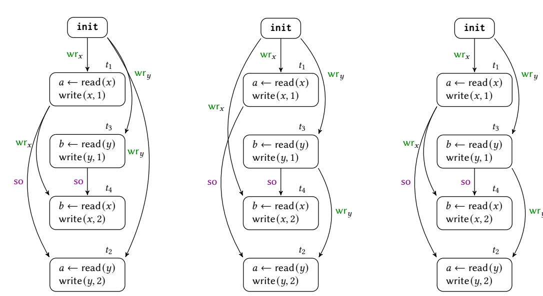
Discuss the exponential total branches ditched

7 THE PARTICULAR CASE OF PREFIX CONSISTENCY

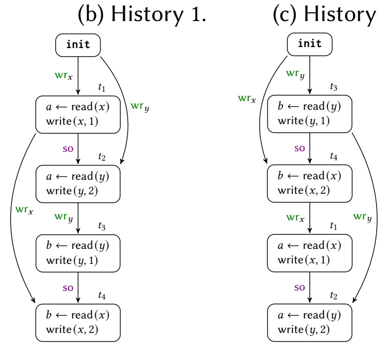
Prefix Consistency is an special isolation level as it is not causal consistent Example but every partial history PRE-consistent can be extended into a consistent history 3.5. Therefore, neither algorithm cite algorithm is guaranteed to be sound, complete and strongly optimal for PRE nor the non-existence of such algorithm can be deduced in the same way as in theorem cite SI and SER dont work. In this section we present one program for which algorithm cite algorithm may or may not be consistent depending on the oracle order employed.

```
begin;           begin;
a=read(x);       a=read(y);
write(x,1);      write(y,2);
commit       ||  commit
begin;           begin;
b=read(y);       b=read(x);
write(y,1);      write(x,2);
commit           commit
```

(a) Program.



(b) History 1. (c) History 2. (d) History 3.



(e) History 4. (f) History 5.

Fig. 10. A program along its PRE-consistent histories.

Let's analyze how algorithm cite algorithm would behave under program depicted in figure 10a; where $<_{or}$ is defined as $t_1 <_{or} t_2 <_{or} t_3 <_{or} t_4$. In this situation, histories h_1 and h_4 (10b and 10e) are computed without any call to SWAP function. It is clear that under h_4's branch no swaps can be produced as transactions are totally ordered via $(\mathsf{so} \cup \mathsf{wr})^*$. However, a swap between t_2 and t_3 can be produced in the history $h_1' = h_1 \setminus t_4$ (which precedes h_1) leading to histories h_2 and h_3 (figure 10c and 10d). However, as the read event in t_2 in both h_2 and h_3 is swapped, we cannot swap t_4 and t_1 by $\textsc{isSwappable}_{\mathcal{M}}$'s definition. Therefore, h_5 (10f) is never reached.

On the other hand, let or' the oracle order defined as $t_1 <_{\mathsf{or}} t_3 <_{\mathsf{or}} t_4 <_{\mathsf{or}} t_2$. In this case, histories h_1, h_2 and h_3 can be obtained without any call to SWAP. Let $h_3' = h_3 \setminus t_2$; the maximum common prefix of both h_1 and h_3. Then, in h_3 (respectively h_3'), every event is maximally added, so t_2 and t_3 can be swapped to obtain h_4 (respectively t_1 and t_4 to obtain h_5). To conclude, with or' as oracle order, the algorithm is complete.

8 PROOF OF THE ALGORITHMS

Introduction

8.1 Soundness

THEOREM 8.1. *Algorithm ?? 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 $\mathsf{so} \cup \mathsf{wr}$-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. □

8.2 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 2 CANONICAL ORDER

procedure CANONICALORDER(h, T, T')
  return T $[\mathsf{so} \cup \mathsf{wr}]^*$ $T' \vee$
    $(\neg(t(T')$ $[\mathsf{so} \cup \mathsf{wr}]^*$ $T) \wedge$ MINIMALDEPENDENCY$(h, T, T', \bot)$
end procedure

procedure MINIMALDEPENDENCY(h, T, T', e)
  let $a = \min_{<_{\mathsf{or}}} \textsc{dep}(h, T, e)$; $a' = \min_{<_{\mathsf{or}}} \textsc{dep}(h, T', e)$
  if $a \neq a'$ **then**
    return $a <_{\mathsf{or}} a'$
  else
    return MINIMALDEPENDENCY(h, T, T', a)
  end if
end procedure

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

The function CANONICALORDER produces a relation between transactions in a history, denoted $\leq^h$. In algorithm 2's description, we denote $\bot$ 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, $\textsc{dep}(h, T, \min_{<_{\mathsf{or}}} \textsc{dep}(h, T, e)) \subseteq \textsc{dep}(h, T, e)$. Moreover, if $\textsc{dep}(h, T, e) \neq T$, the inclusion is strict.*

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

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 $\textsc{dep}(h, T, e)$, 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. $\min_{<_{\mathsf{or}}} \textsc{dep}(h, T, e) \neq \min_{<_{\mathsf{or}}} \textsc{dep}(h, T', e)$. □

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 $c \neq a$, $\neg(c$ $[\mathsf{so} \cup \mathsf{wr}]^*$ $a)$ Here we distinguish four cases:

- If a [so ∪ wr]* b and b [so ∪ wr]* c, then a [so ∪ wr]* c, so $a \leq^h c$.
- If a [so ∪ wr]* b but $\neg(b$ [so ∪ wr]* $c)$, then for every $e \in h$, $\min_{<_{\text{or}}} \textsc{dep}(a, e) \leq_{\text{or}} \min_{<_{\text{or}}} \textsc{dep}(b, e)$, so $a <^h c$.
- If $\neg(a$ [so ∪ wr]* $b)$ but b [so ∪ wr]* c, then for every $e \in h$, $\min_{<_{\text{or}}} \textsc{dep}(b, e) \leq_{\text{or}} \min_{<_{\text{or}}} \textsc{dep}(c, e)$, so $a <^h c$.
- If $\neg(a$ [so ∪ wr]* $b)$ and $\neg(b$ [so ∪ 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 [so ∪ wr]$^*b$, then $a = b$. If not, then $\textsc{minimalDependency}(h, a, b, \perp)$ and $\textsc{minimalDependency}(h, b, a, \perp)$ cannot be satisfied at the same time. Again an induction on $\textsc{minimalDependency}$ along with the history's finiteness.

• Strongly connection Let a, b s.t. $a \not\leq_{\text{or}} b$. If b [so∪wr]$^*a$, then $b \leq_{\text{or}} a$. Otherwise, as $\neg(a$ [so∪wr]* $b)$ and $\textsc{minimalDependency}$ halts (lemma 8.3) and $\neg\textsc{minimalDependency}(h, a, b, e)$, then $\textsc{minimalDependency}(h, b, a, e)$; 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 $\exists e'' \in h, t(e'') \leq_{\text{or}} t(e)$ s.t. $t(e')$ [so ∪ wr]* $t(e''), e'' \leq_h e$ and $\textsc{swapped}(h, e'')$; 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 **??**'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 $e \longleftarrow \textsc{next}(h)$. On one hand, e is not a `read` nor a `begin` event and $h' = h \bullet e$, as $\neg\textsc{swapped}(h, e)$ 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 $d = \texttt{begin}(t(e)), d \in h$. As $R^{\text{or}}(h)$ and $a \notin h$, $a \leq_{\text{or}} d$; so there exists $c \in h, t(c) \leq_{\text{or}} t(a)$ s.t. $t(d)$ [so ∪ wr]* $t(c), c \leq_h d$ and $\textsc{swapped}(h, c)$. As $t(r) = t(d)$, we conclude $R^{\text{or}}(h)$.

Finally, let $h' = \textsc{swap}(h \bullet e, r, w)$ for some $r, w \in h$ s.t. $\textsc{isSwappable}_{\mathcal{M}}(h \bullet e, r, w)$ 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 $\textsc{isSwappable}_{\mathcal{M}}(h \bullet e, r, w)$ 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 $c \in h, t(c) \leq_{\text{or}} t(a) \leq_{\text{or}} t(r)$ s.t. $t(r)$ [so ∪ wr]* $t(c)$ and $\textsc{swapped}(h, c)$. However, this contradicts $\textsc{isSwappable}_{\mathcal{M}}(h \bullet e, r, w)$; so $r \leq_{\text{or}} a$. Taking $e'' = r$ the property is witnessed. □

Proposition 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 ??. 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 $e = \textsc{next}(h)$

- Adding a end, `write`: As h_p and h are edge-wise identical, $\leq^h \equiv \leq_h$.
- Adding a `begin`: As $\textsc{dep}(h_p, T, \bot) = \textsc{dep}(h, T, \bot)$ for every transaction in h_p, if $T \leq^{h_p} T'$, then $T \leq^h T'$. Moreover, $\textsc{dep}(h, t(e), \bot) = \{e\} = \min_{\text{or}} \mathcal{P} \setminus h_p$. By 8.6 h is or-respectful, so for every T, $\min_{\text{or}} \textsc{dep}(h, T, \bot) <_{\text{or}} e$; which implies $T <^h t(e)$. By lemma 8.4, $\leq^h$ is a total order, so it coincides with $\leq^h$.
- Adding a `read`: As no transaction depends on $t(e)$ and $t(e) = \texttt{last}(h_p)$, if we prove that for every pair of transactions $\textsc{minimalDependency}(h_p, T, T', \bot) = \textsc{minimalDependency}(h, T, T', \bot)$, the lemma would hold. On one hand, $\textsc{dep}(h, t(e), \bot) = \textsc{dep}(h_p, t(e), \bot) = t(e)$ and in the other hand, by lemma 8.6, $\min_{\text{or}} \textsc{dep}(h_p, T, \bot) <_{\text{or}} t(e)$. Finally, as $e \notin \textsc{dep}(h, T, e')$, for every $T \neq t(e), e' \neq \bot$, for every pair of transactions T, T', $\textsc{minimalDependency}(h_p, T, T' \bot) = \textsc{minimalDependency}(h, T, T' \bot)$.
- Swapping $r \in h$ and $w \in t(e)$: As $\textsc{isSwappable}_{\mathcal{M}}(h \bullet e, r, w)$ is satisfied and h is or-respectful, for every event e' and transaction T, $\min_{\text{or}} \textsc{dep}(h_p, T, e') = \min_{\text{or}} \textsc{dep}(h, T, e')$, so for every pair of transactions $\textsc{minimalDependency}(h_p, T, T', \bot) = \textsc{minimalDependency}(h, T, T', \bot)$. In particular, this implies $T \leq^{h_p} T'$ if and only if $T \leq^h T'$ for every pair T, T' and $T \leq^h t(r)$; so $\leq^h \equiv \leq_h$.

□

Proposition 8.7 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 8.7 together with lemma 8.6 justify enlarging definition 8.5 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 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. $t(c) \leq_{\text{or}} t(e)$, $t(e')$ $[\text{so} \cup \text{wr}]^* t(c)$, $\textsc{swapped}(h, c)$ and $c \leq^h e$; so the property is satisfied. Otherwise, by definition of $\textsc{minimalDependency}$, there exists $r' \in h$ s.t. T' $[\text{so} \cup \text{wr}]^*$ $t(r')$ and $t(r') \leq_{\text{or}} T$. Moreover, by $\textsc{canonicalOrder}$'s definition, $t(r) \leq^h T$. Finally $\textsc{swapped}(h, r')$ holds as it is the minimum element according or. To sum up, $R^{\text{or}}(h)$ holds. □

Algorithm 3 PREV

procedure PREV(h)
  if $h = \emptyset$ **then**
    return $\emptyset$
  end if
  $a \leftarrow \texttt{last}(h)$
  if $\neg$SWAPPED(h, a) **then**
    return $h \setminus a$
  else
    return MAXCOMPLETION$(h \setminus a, \{e \mid e \notin (h \setminus a) \wedge e <_{\mathsf{or}} h.\mathsf{wr}(a)\})$
  end if
end procedure

procedure MAXCOMPLETION(h, D)
  if $D \neq \emptyset$ **then**
    $e \leftarrow \min_{<_{\mathsf{or}}} D$
    if $e.type() \neq \texttt{read}$ **then**
      return MAXCOMPLETION$(h \bullet e, D \setminus \{e\})$
    else
      let w s.t. $\text{readsLatest}_I(h \bullet_w e, e,)$
      return MAXCOMPLETION$(h \bullet_w e, D \setminus \{e\})$
    end if
  else
    return h
  end if
end procedure

Function 3 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 8.9. *For every* or*-respectful history h,* PREV(h) *is also* or*-respectful.*

PROOF. Let suppose $h \neq \emptyset$, $h_p =$ PREV(h), $a = \texttt{last}(h)$, $e \in \mathcal{P}$ 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 $\exists e'' \in h, t(e'') \leq_{\mathsf{or}} t(e), e'' \leq^h e, t(e')\ [\mathsf{so} \cup \mathsf{wr}]^* t(e'')$ 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, SWAPPED(h, a) and we distinguish between the sets e and e' belong to. Firstly, for every pair of events $\hat{e} \in h_p \setminus h$, $\hat{e}' \in$ DEP$(h, t(\hat{e}, \bot))$, we know that $t(\hat{e}) \leq_{\mathsf{or}} t(\hat{e}')$. Therefore, $\min_{<_{\mathsf{or}}}$ DEP$(h, t(\hat{e}, \bot)) = \texttt{begin}(t(\hat{e}))$. In addition, by construction of PREV(h) and or-respectfulness of h, for every $\hat{e} \in h, e'' \in h$, $\min_{<_{\mathsf{or}}}$ DEP$(h_p, t(\hat{e}), e'') = \min_{<_{\mathsf{or}}}$ DEP$(h, t(\hat{e}), e'')$. 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 $R^{\mathsf{or}}(h)$ 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

$e' \notin h$, $\neg(t(e')\ [\mathsf{so} \cup \mathsf{wr}]^*\ t(e))$, so regardless if $t(e)\ [\mathsf{so} \cup \mathsf{wr}]^* t(e')$, $e \leq^{h_p} e'$. To sum up, $R^{\mathsf{or}}(h_p)$ holds. □

Lemma 8.10. *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, $h_n = h_p \bullet_{h.\mathsf{wr}(a)} a$ 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 $w = h.\mathsf{wr}(a)$. Firstly, as $\textsc{swapped}(h, a)$, $a <_{\mathsf{or}} w$, and by lemma 8.6, $R^{\mathsf{or}}(h_p)$ 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 $e <_{\mathsf{or}} w$. In particular, $\neg(t(e)\ [\mathsf{so} \cup \mathsf{wr}]^*\ t(w))$. Moreover, if $e \leq_{\mathsf{or}} a$, by $R^{\mathsf{or}}(h)$, either $e \leq^h a$ or $\exists e'' \in h, e'' \leq_{\mathsf{or}} e$ s.t. $t(a)\ [\mathsf{so} \cup \mathsf{wr}]^* t(e'')$, $e'' \leq^h e$ and $\textsc{swapped}(h, e'')$; both impossible situations as $e \notin h$ and $a = \mathtt{last}(h)$; so $a \leq_{\mathsf{or}} e$. 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 $\neg(t(e)\ [\mathsf{so} \cup \mathsf{wr}]^*\ t(w))$ and $a <_{\mathsf{or}} e <_{\mathsf{or}} w$. If $e \in h$ then $e \leq^h a$, and as h is or-respectful and $a \leq_{\mathsf{or}} e$, we deduce there exists a $e'' \in h$ s.t. $t(e'') \leq_{\mathsf{or}} t(a)$, $t(e)\ [\mathsf{so} \cup \mathsf{wr}]^* t(e'')$ 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 8.11. *In a consistent* or*-respectful history h whose previous history is reachable, if its last event a is swapped, h coincides with* $\textsc{swap}(\textsc{prev}(h), a, h.\mathsf{wr}(a))$.

Proof. It comes straight away from the proof of lemma 8.10. □

Lemma 8.12. *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 $\textsc{maxCompletion}$. Let us call $D = \{e \mid e \notin h' \wedge e <_{\mathsf{or}}\}$. 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 $D_k = D_{k+1} \setminus \{\min_{<_{\mathsf{or}}} D_{k+1}\}$, $e_k = \min_{<_{\mathsf{or}}} D_k$ 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 $\textsc{isConsistent}_{\mathcal{M}}(h_{k+1} \bullet_{f_{k+1}} e_{k+1}) \wedge t(f_{k+1})\ [\mathsf{so} \cup \mathsf{wr}]^*\ t(e_{k+1})$ 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 $E_{k+1} = \{w \mid \textsc{isConsistent}_{\mathcal{M}}(h_{k+1} \bullet_w e_{k+1}) \wedge$

$\{e \mid \text{SWAPPED}(h_{k+1}, e)\} = \{e \mid \text{SWAPPED}(h_{k+1} \bullet_w e_{k+1}, e)\}\}$ 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 $\{e \mid \text{SWAPPED}(h_k, e)\} = \{e \mid \text{SWAPPED}(h', e)\}$. 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 $\text{readsLatest}_I(h_k, e_{k+1}, w_{k+1})$. Altogether, we obtain $h_p = h_0$; which let us conclude $\{e \in h_p \mid \text{SWAPPED}(h_p, e)\} = \{e \in h' \mid \text{SWAPPED}(h', e)\} = \{e \in h \mid \text{SWAPPED}(h, e)\} \setminus \{a\}$. □

Lemma 8.13. *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 8.12. 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 $\xi(x_p) = \xi(h)$ 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 $\text{PREV}^{k_{h_p}}(h_p) = \emptyset$. Therefore, $k_h = k_{h_p} + 1$ satisfies $\text{PREV}^{k_h}(h) = \emptyset$. □

Proposition 8.14. *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 $\text{PREV}^k(h) = \emptyset$, which exists thanks to lemma 8.13 and $C = \{\text{PREV}^{k-n}(h)\}_{n=0}^{k}$ a set of indexed histories. By the collection's definition and lemma 8.9, $h_0 = \text{PREV}^k(h) = \emptyset$, $h_k = \text{PREV}^0(h) = h$ 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 8.10, we know that if h_n is reachable, h_{n+1} is it too; which proves the inductive step. □

Theorem 8.15. *Algorithm ?? is complete.*

Proof. By lemma 8.8, any consistent total history is or-respectful. As a consequence of proposition 8.14, there exist a sequence of reachable histories which h belongs to; so in particular, h is reachable. □

8.3 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!

Lemma 8.16. *For every two histories h_1, h_2 with the same set of events and `read` event r, if $\text{ISSWAPPABLE}_{\mathcal{M}}(h_i, r, w)$ holds for $i \in \{1, 2\}$ and $\leq_{h_1} \equiv \leq_{h_2}$, for every `read` event r' s.t. $r \leq_{h_i} r'$ and $\neg(t(r')\ [\text{so} \cup \text{wr}]^+ t(w))$ we have $h_1.\text{wr}(r') = h_2.\text{wr}(r')$.*

Proof. As for every $i \in \{1, 2\}$ $\text{ISSWAPPABLE}_{\mathcal{M}}(h_1, r, w)$ holds, so does it $\text{readsLatest}_I(h_i, r')$. Therefore, as $\leq_{h_1} \equiv \leq_{h_2}$, $h_1.\text{wr}(r') = h_2.\text{wr}(r')$. □

Theorem 8.17. *Algorithm ?? 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 8.7, 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 $\textsc{next}$ function $h_2 = h \setminus \texttt{last}(h) = h_1$. On the contrary, let's suppose $r = \texttt{last}(h)$ is a swapped event that reads from a `write` event w. 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 8.11, $h = \textsc{swap}(h_1, r, w)$, so $h_1 \restriction_{h \setminus r} = h_2 \restriction_{h \setminus r}$. As h_1, h_2 are both or-respectful, $e \in h_1 \setminus h \iff e \in h_2 \setminus h$. Finally, as $\textsc{isSwappable}_{\mathcal{M}}(h_i, r, w)$ holds for $i \in \{1, 2\}$, by lemma 8.16, if $e \in h_1 \setminus h$, $h_1.\mathsf{wr}(e) = h_2.\mathsf{wr}(e)$. Therefore, $h_1 = h_2$. □