Dynamic Partial Order Reduction for Checking Correctness Against Transaction Isolation Levels

AHMED BOUAJJANI, Université Paris Cité, CNRS, IRIF, France
CONSTANTIN ENEA, LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris, France
ENRIQUE ROMÁN-CALVO, Université Paris Cité, CNRS, IRIF, 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 isolation for performance. The weaker the isolation level is, 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 stateless model checking algorithms for studying correctness of such applications that rely on dynamic partial order reduction. These algorithms work for a number of widely-used weak isolation levels, including Read Committed, Causal Consistency, Snapshot Isolation, and Serializability. We show that they are complete, sound and optimal, and run with polynomial memory consumption in all cases. We report on an implementation of these algorithms in the context of Java Pathfinder applied to a number of challenging applications drawn from the literature of distributed systems and databases.

CCS Concepts: • Theory of computation → Verification by model checking; Distributed computing models; • Software and its engineering → Formal software verification.

Additional Key Words and Phrases: Applications of Storage Systems, Transactional Databases, Weak Isolation Levels, Dynamic Partial-Order Reduction

ACM Reference Format:

1 INTRODUCTION

Data storage is no longer about writing data to a single disk with a single point of access. Modern applications require not just data reliability, but also high-throughput concurrent accesses. Applications concerning supply chains, banking, etc. use traditional relational databases for storing and processing data, whereas applications such as social networking software and e-commerce platforms use cloud-based storage systems (such as Azure Cosmos DB [Paz 2018], Amazon DynamoDB [DeCandia et al. 2007], Facebook TAO [Bronson et al. 2013], etc.).

Providing high-throughput processing, unfortunately, comes at an unavoidable cost of weakening the consistency guarantees offered to users: Concurrently-connected clients may end up observing different versions of the same data. These “anomalies” can be prevented by using a strong isolation level such as Serializability [Papadimitriou 1979], which essentially offers a single version of the

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2023 Copyright held by the owner/author(s).
2475-1421/2023/6-ART129
https://doi.org/10.1145/3591243
data to all clients at any point in time. However, serializability requires expensive synchronization and incurs a high performance cost. As a consequence, most storage systems use weaker isolation levels, such as \textit{Causal Consistency} [Akkoorath and Bieniusa 2016; Lamport 1978; Lloyd et al. 2011], \textit{Snapshot Isolation} [Berenson et al. 1995], \textit{Read Committed} [Berenson et al. 1995], etc. for better performance. In a recent survey of database administrators [Pavlo 2017], 86% of the participants responded that most or all of the transactions in their databases execute at Read Committed level.

A weaker isolation level allows for more possible behaviors than stronger isolation levels. It is up to the developers then to ensure that their application can tolerate this larger set of behaviors. Unfortunately, weak isolation levels are hard to understand or reason about [Adya 1999; Brutschy et al. 2017] and resulting application bugs can cause loss of business [Warszawski and Bailis 2017].

\textbf{Model Checking Database-Backed Applications.} This paper addresses the problem of model checking code for correctness against a given isolation level. Model checking [Clarke et al. 1983; Queille and Sifakis 1982] explores the state space of a given program in a systematic manner and it provides high coverage of program behavior. However, it faces the infamous state explosion problem, i.e., the number of executions grows exponentially in the number of concurrent clients.

\textit{Partial order reduction} (POR) [Clarke et al. 1999; Godefroid 1996; Peled 1993; Valmari 1989] is an approach that limits the number of explored executions without sacrificing coverage. POR relies on an equivalence relation between executions where e.g., two executions are equivalent if one can be obtained from the other by swapping consecutive independent (non-conflicting) execution steps. It guarantees that at least one execution from each equivalence class is explored. \textit{Optimal} POR techniques explore exactly one execution from each equivalence class. Beyond this classic notion of optimality, POR techniques may aim for optimality by avoiding visiting states from which the exploration is blocked. \textit{Dynamic} partial order reduction (DPOR) [Flanagan and Godefroid 2005] has been introduced to explore the execution space (and tracking the equivalence relation between executions) on-the-fly without relying on a-priori static analyses. This is typically coupled with \textit{stateless} model checking (SMC) [Godefroid 1997] which explores executions of a program without storing visited states, thereby, avoiding excessive memory consumption.

There is a large body of work on (D)POR techniques that address their soundness when checking a certain class of specifications for a certain class of programs, as well as their completeness and their theoretical optimality (see Section 8). Most often these works consider shared memory concurrent programs executing under a strongly consistent memory model.

In the last few years, some works have studied DPOR in the case of shared memory programs running under weak memory models such as TSO or Release-Acquire, e.g. [Abdulla et al. 2017a, 2016, 2018; Kokologiannakis et al. 2019]. While these algorithms are sound and complete, they have exponential space complexity when they are optimal. More recently, Kokologiannakis et al. [2022] defined a DPOR algorithm that has a polynomial space complexity, in addition of being sound, complete and optimal. This algorithm can be applied for a range of shared memory models.

While the works mentioned above concern shared memory programs, we are not aware of any published work addressing the case of database transactional programs running under weak isolation levels. In this paper, we address this case and propose new stateless model checking algorithms relying on DPOR techniques for database-backed applications. We assume that all the transactions in an application execute under the same isolation level, which happens quite frequently in practice (as mentioned above, most database applications are run on the default isolation level of the database). Our work generalizes the approach introduced by [Kokologiannakis et al. 2022]. However, this generalization to the transactional case, covering the most relevant isolation levels, is not a straightforward adaptation of [Kokologiannakis et al. 2022]. Ensuring
optimality while preserving the other properties, e.g., completeness and polynomial memory complexity, is very challenging. Next, we explain the main steps and features of our work.

**Formalizing Isolation Levels.** Our algorithms rely on the axiomatic definitions of isolation levels introduced by Biswas and Enea [2019]. These definitions use logical constraints called *axioms* to characterize the set of executions of a database (e.g., key-value store) that conform to a particular isolation level (extensible to SQL queries [Biswas et al. 2021]). These constraints refer to a specific set of relations between events/transactions in an execution that describe control-flow or data-flow dependencies: a program order $\text{po}$ between events in the same transaction, a session order $\text{so}$ between transactions in the same session, and a write-read $\text{wr}$ (read-from) relation that associates each read event with a transaction that writes the value returned by the read. These relations along with the events in an execution are called a *history*. A history describes only the interaction with the database, omitting application-side events (e.g., computing values written to the database).

**Execution Equivalence.** DPOR algorithms are parametrized by an equivalence relation on executions, most often, Mazurkiewicz equivalence [Mazurkiewicz 1986]. In this work, we consider a weaker equivalence relation, also known as *read-from equivalence* [Abdulla et al. 2019, 2018; Chalupa et al. 2018; Kokologiannakis et al. 2022, 2019; Kokologiannakis and Vafeiadis 2020], which considers that two executions are equivalent when their histories are precisely the same (they contain the same set of events, and the relations $\text{po}$, $\text{so}$, and $\text{wr}$ are the same). In general, read-from equivalence is coarser than Mazurkiewicz equivalence, and its equivalence classes can be exponentially-smaller than Mazurkiewicz traces in certain cases [Chalupa et al. 2018].

**SMC Algorithms.** Our SMC algorithms enumerate executions of a given program under a given isolation level $I$. They are *sound*, i.e., enumerate only *feasible* executions (admitted by the program under $I$), *complete*, i.e., they output a representative of each read-from equivalence class, and *optimal*, i.e., they output exactly one complete execution from each read-from equivalence class. For isolation levels weaker than and including Causal Consistency, they satisfy a notion of *strong optimality* which says that additionally, the enumeration avoids states from which the execution is “blocked”, i.e., it cannot be extended to a complete execution of the program. For Snapshot Isolation and Serializability, we show that *there exists* no algorithm in the same class (to be discussed below) that can ensure such a strong notion of optimality. All the algorithms that we propose are polynomial space, as opposed to many DPOR algorithms introduced in the literature.

As a starting point, we define a generic class of SMC algorithms, called *swapping based*, generalizing the approach adopted by [Kokologiannakis et al. 2022, 2019], which enumerate histories of program executions. These algorithms focus on the interaction with the database assuming that the other steps in a transaction concern local variables visible only within the scope of the enclosing session. Executions are extended according to a generic scheduler function $\text{Next}$ and every read event produces several exploration branches, one for every write executed in the past that it can read from. Events in an execution can be swapped to produce new exploration “roots” that lead to different histories. Swapping events is required for completeness, to enumerate histories where a read $r$ reads from a write $w$ that is scheduled by $\text{Next}$ after $r$. To ensure soundness, we restrict the definition of swapping so that it produces a history that is feasible by construction (extending an execution which is possibly infeasible may violate soundness). Such an algorithm is optimal w.r.t. the read-from equivalence when it enumerates each history exactly once.

We define a concrete algorithm in this class that in particular, satisfies the stronger notion of optimality mentioned above for every isolation level $I$ which is *prefix-closed* and *causally-extensible*, e.g., *Read Committed* and *Causal Consistency*. Prefix-closure means that every prefix of a history that satisfies $I$, i.e., a subset of transactions and all their predecessors in the causal relation, i.e.,

---

1 A session is a sequential interface to the storage system. It corresponds to what is also called a *connection*. 

\[ x \in \text{Vars} \quad a \in \text{LVars} \]

\[
\begin{align*}
\text{Prog} & ::= \text{Sess} \mid \text{Sess} \mid | \mid \text{Prog} \\
\text{Sess} & ::= \text{Trans} \mid \text{Trans} \mid \text{Sess} \\
\text{Trans} & ::= \text{begin} \text{Body} \mid \text{commit} \\
\text{Body} & ::= \text{Instr} \mid \text{Instr} \text{Body} \\
\text{Instr} & ::= \text{InstrDB} \mid a := e \mid \text{if}(\phi(\vec{a}))\{\text{Instr}\} \\
\text{InstrDB} & ::= a := \text{read}(x) \mid \text{write}(x,a) \mid \text{abort}
\end{align*}
\]

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

\((\text{so} \cup \text{wr})^+\), is also consistent with \( I \), and causal extensibility means that any pending transaction in a history that satisfies \( I \) can be extended with one more event to still satisfy \( I \), and if this is a read event, then, it can read-from a transaction that precedes it in the causal relation. To ensure strong optimality, this algorithm uses a carefully chosen condition for restricting the application of event swaps, which makes the proof of completeness in particular, quite non-trivial.

We show that isolation levels such as Snapshot Isolation and Serializability are not causally-extensible and that there exists no swapping based SMC algorithm which is sound, complete, and strongly optimal at the same time (independent of memory consumption bounds). This impossibility proof uses a program to show that any NEXT scheduler and any restriction on swaps would violate either completeness or strong optimality. However, we define an extension of the previous algorithm which satisfies the weaker notion of optimality, while preserving soundness, completeness, and polynomial space complexity. This algorithm will simply enumerate executions according to a weaker prefix-closed and causally-extensible isolation level, and filter executions according to the stronger isolation levels Snapshot Isolation and Serializability at the end, before outputting.

We implemented these algorithms in the Java Pathfinder (JPF) model checker [Visser et al. 2004], and evaluated them on a number of challenging database-backed applications drawn from the literature of distributed systems and databases.

Our contributions and outline are summarized as follows:

§ 3 identifies a class of isolation levels called prefix-closed and causally-extensible that admit efficient SMC.

§ 4 defines a generic class of swapping based SMC algorithms based on DPOR which are parametrized by a given isolation level.

§ 5 defines a swapping based SMC algorithm which is sound, complete, strongly-optimal, and polynomial space, for any isolation level that is prefix-closed and causally-extensible.

§ 6 shows that there exists no swapping based algorithm for Snapshot Isolation and Serializability, which is sound, complete, and strongly-optimal at the same time, and proposes a swapping based algorithm which satisfies “plain” optimality.

§ 7 reports on an implementation and evaluation of these algorithms.

Section 2 recalls the formalization of isolation levels of Biswas and Enea [Biswas and Enea 2019; Biswas et al. 2021], while Sections 8 and 9 conclude with a discussion of related work and concluding remarks. Additional formalization, proofs, and experimental data can be found in the technical report [Bouajjani et al. 2023].

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 either commit or abort instructions, and its body contains instructions that access the database and manipulate a set LVars of local variables. We use symbols \( a, b \), etc. to denote elements of LVars.
For simplicity, we abstract the database state as a valuation to a set \( \text{Vars} \) of global variables\(^2\), ranged over using \( x, y, \text{etc.} \). The instructions accessing the database correspond to reading the value of a global variable and storing it into a local variable \( a (a := \text{read}(x)) \), writing the value of a local variable \( a \) to a global variable \( x (\text{write}(x, a)) \), 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 \( \phi(a) \) over a set of local variables \( a \) (their syntax is not important). Our results assume bounded programs, as usual in SMC algorithms, and therefore, we omit other constructs like while loops. SQL statements (SELECT, JOIN, UPDATE) manipulating relational tables can be compiled to reads or writes of variables representing rows in a table (see for instance, [Biswas et al. 2021; Rahmani et al. 2019]).

### 2.2 Isolation Levels

We present the axiomatic framework introduced by Biswas and Enea [2019] for defining isolation levels. 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 an execution.

#### 2.2.1 Histories

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

A transaction log \( \langle t, E, \text{po}_t \rangle \) is an identifier \( t \) and a finite set of events \( E \) along with a strict total order \( \text{po}_t \) on \( E \), called program order (representing the order between instructions in the body of a transaction). The minimal element of \( \text{po}_t \) is a begin event. A transaction log without neither a commit nor an abort event is called pending. Otherwise, it is called complete. A complete transaction log with a commit event is called committed and aborted otherwise. If a commit or an abort event occurs, then it is maximal in \( \text{po}_t \); commit and abort cannot occur in the same log. The set \( E \) of events in a transaction log \( t \) is denoted by \( \text{events}(t) \). Note that a transaction is aborted because it executed an \texttt{abort} instruction. Histories do not include transactions aborted by the database because their effect should not be visible to other transactions and the abort is not under the control of the program. For simplicity, we may use the term transaction instead of transaction log.

Isolation levels differ in the values returned by read events which are not preceded by a write on the same variable in the same transaction. We assume in the following that every transaction in a program is executed under the same isolation level. For every isolation level that we are aware of, if a read of a global variable \( x \) is preceded by a write to \( x \) in \( \text{po}_t \), then it should return the value written by the last write to \( x \) before the read (w.r.t. \( \text{po}_t \)).

The set of \texttt{read}(\( x \)) events in a transaction log \( t \) that are not preceded by a write to \( x \) in \( \text{po}_t \), for some \( x \), is denoted by \( \text{reads}(t) \). Also, if \( t \) does not contain an abort event, the set of \texttt{write}(\( x, _ \)) events in \( t \) that are not followed by other writes to \( x \) in \( \text{po}_t \), for some \( x \), is denoted by \( \text{writes}(t) \). If a transaction contains multiple writes to the same variable, then only the last one (w.r.t. \( \text{po}_t \)) can be visible to other transactions (w.r.t. any isolation level that we are aware of). If \( t \) contains an abort event, then we define \( \text{writes}(t) \) to be the empty set. This is because the effect of aborted transactions (its set of writes) should not be visible to other transactions. The extension to sets of transaction logs is defined as usual. Also, we say that a transaction log \( t \) \text{writes} \( x \), denoted by \( t \text{ writes } x \), when \( \text{writes}(t) \) contains some \texttt{write}(\( x, _ \)) event.

\(^2\)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.
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. It also includes a write-read relation (also called read-from) that defines read values by associating each read to a transaction that wrote that value. Read events do not contain a value, and their return value is defined as the value written by the transaction associated by the write-read relation. Let be a set of transaction logs. For a write-read relation so writes(T) and variable x, wrx is the restriction of wr to reads of x, wrx = wr \cap (writes(T) \times \{e \mid e is a read(x) event\}). We extend the relations wr and wrx to pairs of transactions by \langle t_1, t_2 \rangle \in wr, resp., \langle t_1, t_2 \rangle \in wrx, iff there exists a write(x, ) event w in t_1 and a read(x) event r in t_2 s.t. \langle w, r \rangle \in wr, resp., \langle w, r \rangle \in wrx. Analogously, wr and wrx can be extended to tuples formed of a transaction (containing a write) and a read event. We say that the transaction log t_1 is read by the transaction log t_2 when \langle t_1, t_2 \rangle \in wr.

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

- the inverse of so is a total function,
- if \langle w, r \rangle \in wr, then w and r are a write and respectively, a read, of the same variable, and
- so \cup wr is acyclic (here we use the extension of wr to pairs of transactions).

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

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

We extend so to pairs of events by \langle e_1, e_2 \rangle \in so if \langle tr(h, e_1), tr(h, e_2) \rangle \in so. Also, po = \bigcup_{t \in T} po_t.

**2.2.2 Axiomatic Framework.** A history satisfies a certain isolation level if there is a strict total order co on its transactions, called commit order, which extends the write-read relation and the session order, and which satisfies certain properties. These properties, called axioms, relate the commit order with the so and wr relations in a history and are defined as first-order formulas of the form:

\[ \forall x, \forall t_1 \neq t_2, \forall t_3. \]

\[ \langle t_1, t_3 \rangle \in wrx \land t_2 writes x \land \phi(t_2, t_3) \Rightarrow \langle t_2, t_1 \rangle \in co \]  

where \phi is a property relating t_2 and r (i.e., the read or the transaction reading from t_1) that varies from one axiom to another.\(^3\) Note that an aborted transaction t cannot take the role of t_1 nor

These formulas are interpreted on tuples \langle h, co \rangle of a history h and a commit order co on the transactions in h as usual.
We define a small-step operational semantics for transactional programs, which is parametrized in equation 1 as the set \( \text{writes}(t) \) is empty. Intuitively, this axiom schema states the following: in order for \( \tau \) to read specifically \( t_1 \)’s write on \( k \), it must be the case that every \( t_2 \) that also writes \( k \) and satisfies \( \phi(t_2, \tau) \) was committed before \( t_1 \). The property \( \phi \) relates \( t_2 \) and \( \tau \) using the relations in a history and the commit order. Figure 2 shows two axioms which correspond to their homonymous isolation levels: Causal Consistency (CC) and Serializability (SER). The conjunction of the other two axioms Conflict and Prefix defines Snapshot Isolation (SI). Read Atomic (RA) is a weakening of CC where \((\text{so} \cup \text{wr})^+\) is replaced with \(\text{so} \cup \text{wr}\). Read Committed (RC) is defined similarly. Note that SER is stronger than SI (i.e., every history satisfying SER satisfies SI as well), SI is stronger than CC, CC is stronger than RA, and RA is stronger than RC.

For instance, the axiom defining Causal Consistency [Lamport 1978] states that for any transaction \( t_1 \) writing a variable \( x \) that is read in a transaction \( t_3 \), the set of \((\text{wr} \cup \text{so})^+\) predecessors of \( t_3 \) writing \( x \) must precede \( t_1 \) in commit order (\((\text{wr} \cup \text{so})^+\) is usually called the causal order). A violation of this axiom can be found in Figure 3: the transaction \( t_2 \) writing 2 to \( x \) is a \((\text{wr} \cup \text{so})^+\) predecessor of the transaction \( t_3 \) reading 1 from \( x \) because the transaction \( t_4 \), writing 1 to \( y \), reads \( x \) from \( t_2 \) and \( t_3 \) reads \( y \) from \( t_4 \). This implies that \( t_2 \) should precede in commit order the transaction \( t_1 \) writing 1 to \( x \), which is inconsistent with the write-read relation (\( t_2 \) reads from \( t_1 \)).

The Serializability axiom requires that for any transaction \( t_1 \) writing to a variable \( x \) that is read in a transaction \( t_3 \), the set of \( \text{co} \) predecessors of \( t_3 \) writing \( x \) must precede \( t_1 \) in commit order. This ensures that each transaction observes the effects of all the \( \text{co} \) predecessors.

**Definition 2.2.** For an isolation level \( I \) defined by a set of axioms \( X \), a history \( h = \langle T, \text{so}, \text{wr} \rangle \) satisfies \( I \) iff there is a strict total order \( \text{co} \) s.t. \( \text{wr} \cup \text{so} \subseteq \text{co} \) and \( \langle h, \text{co} \rangle \) satisfies \( X \).

A history that satisfies an isolation level \( I \) is called \( I \)-consistent. For two isolation levels \( I_1 \) and \( I_2 \), \( I_1 \) is weaker than \( I_2 \) when every \( I_1 \)-consistent history is also \( I_2 \)-consistent.

### 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 consistency with \( I \).

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

The operational semantics is defined as a transition relation \( \Rightarrow_t \) between configurations, which are defined as tuples containing the following:

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

The relation \( \Rightarrow_t \) is defined using a set of rules as expected. Starting a new transaction in a session \( j \) is enabled as long as this session has no live transactions (\( \tilde{B}(j) = \epsilon \)) and results in

---

We define two properties of isolation levels, prefix-closure and causal extensibility, which enable the construction of efficient DPOR algorithms (as shown in Section 5).

**Fig. 4.** Explaining the notion of prefix of a history. *init* denotes the transaction log writing initial values. Boxes group events from the same transaction.

Adding a transaction log with a single begin event to the history and scheduling the body of the transaction (adding it to \( B(j) \)). Local steps, i.e., checking a Boolean condition or computation with local variables, use the local variable valuations and advance the code as expected. Read instructions of some global variable \( x \) can have two possible behaviors: (1) if the read follows a write on \( x \) in the same transaction, then it returns the value written by the last write on \( x \) in that transaction, and (2) otherwise, the read reads from another transaction \( t' \) which 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 \( I \). Depending on the isolation level, there may not exist a transaction \( t' \) the read can read from. For other instructions, e.g., commit and abort, the history is simply extended with the corresponding events while ending the transaction execution in the case of aborts.

An initial configuration for program \( P \) contains the program \( P \), a history \( h = \langle \{ t_0 \}, \emptyset, \emptyset \rangle \) where \( t_0 \) is a transaction log containing writes that write the initial value for all variables, and empty current transaction code (\( B = e \)). An execution of a program \( P \) under an isolation level \( I \) is a sequence of configurations \( c_0c_1 \ldots c_n \) 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 (\( P = \emptyset \)). Let \( \text{hist}_I(P) \) 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

We define two properties of isolation levels, prefix-closure and causal extensibility, which enable efficient DPOR algorithms (as shown in Section 5).

## 3.1 Prefix Closure

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 \( \langle t, E, po_I \rangle \) is a transaction log \( \langle t, E', po_I \downarrow E' \times E' \rangle \) such that \( E' \) is \( po_I \)-downward closed. A prefix of a history \( h = \langle T, so, wr \rangle \) is a history \( h' = \langle T', so \downarrow T' \times T', wr \downarrow T' \times T' \rangle \) such that every transaction log in \( T' \) is a prefix of a different transaction log in \( T \) but carrying the same id, \( \text{events}(h') \subseteq \text{events}(h) \), and \( \text{events}(h') \) is \( (po \cup so \cup wr)^* \)-downward closed. For example, the history pictured in Fig. 4b is a prefix of the one in Fig. 4a while the history in Fig. 4c is not. The transactions on the bottom of Fig. 4c have a \( wr \) predecessor in Fig. 4a which is not included.

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

Every isolation level \( I \) discussed above is prefix-closed because if a history \( h \) is \( I \)-consistent with a commit order \( co \), then the restriction of \( co \) to the transactions that occur in a prefix \( h' \) of \( h \) satisfies the corresponding axiom(s) when interpreted over \( h' \).

**Theorem 3.2.** Read Committed, Read Atomic, Causal Consistency, Snapshot Isolation, and Serializability are prefix closed.
We start with an example to explain causal extensibility. Let us consider the histories that will require that at least the \((so \cup wr)\) of \(t\) is maximal in \(t\). These histories satisfy Read Atomic. The history \(t\) does not contain any transaction \(x\) that writes \(y\) while still preserving consistency. The restriction to \(wr\) on the right in Figure 5b which is a \(so\) edge from \(init\) to the other transactions are omitted for legibility.

3.2 Causal Extensibility

We start with an example to explain causal extensibility. Let us consider the histories \(h_1\) and \(h_2\) in Figures 5a and 5b, respectively, without the events \(read(y)\) and \(write(y, 2)\) written in blue bold font. These histories satisfy Read Atomic. The history \(h_1\) can be extended by adding the event \(read(y)\) and the \(wr\) dependency \(wr(init, read(y))\) while still satisfying Read Atomic. On the other hand, the history \(h_2\) can not be extended with the event \(write(y, 2)\) while still satisfying Read Atomic. Intuitively, if the reading transaction on the bottom reads \(x\) from the transaction on the right, then it should read \(y\) from the same transaction because this is more “recent” than \(init\) w.r.t. session order. The essential difference between these two extensions is that the first concerns a transaction which is maximal in \((so \cup wr)^+\) while the second no. The extension of \(h_2\) concerns the transaction on the right in Figure 5b which is a \(wr\) predecessor of the reading transaction. Causal extensibility will require that at least the \((so \cup wr)^+\) maximal (pending) transactions can always be extended with any event while still preserving consistency. The restriction to \((so \cup wr)^+\) maximal transactions is intuitively related to the fact that transactions should not read from non-committed (pending) transactions, e.g., the reading transaction in \(h_2\) should not read from the still pending transaction that writes \(x\) and later \(y\).

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

- \(e\) is added to \(t\) as a maximal element of \(po\),
- if \(e\) is a read event and \(t\) does not contain a write to \(var(e)\), then \(wr\) is extended with some tuple \((t', e)\) such that \((t', t) \in (so \cup wr)^+\) in \(h\) if \(e\) is a read event and \(t\) does contain a write to \(var(e)\), then the value returned by \(e\) is the value written by the latest write on \(var(e)\) before \(e\) in \(t\); the definition of the return value in this case is unique and does not involve \(wr\) dependencies,
- the other elements of \(h\) remain unchanged in \(h'\).

For example, Figure 7b and 7c present two causal extensions with a \(read(x)\) event of the transaction \(t_4\) in the history \(h\) in Figure 7a. The new read event reads from transaction \(t_1\) or \(t_3\) which were...
We define a class of stateless model checking algorithms for enumerating executions of a given history \( h \). For event \( e \) (as in § 2.3), every \( \text{(so \cup wr)} \)-maximal pending transaction \( t \) in \( h \), and every event \( e \), there exists a causal extension \( h' \) of \( t \) with \( e \) that is \( I \)-consistent.

**Definition 3.3.** An isolation level \( I \) is called **causally-extensible** if for every \( I \)-consistent history \( h \), every \( \text{(so \cup wr)} \)-maximal pending transaction \( t \) in \( h \), and every event \( e \), there exists a causal extension \( h' \) of \( t \) with \( e \) that is \( I \)-consistent.

**Theorem 3.4.** Causal Consistency, Read Atomic, and Read Committed are causally-extensible.

Concerning notations, let \( \text{wr} \) along with the valuation of local variables in the transaction of \( e \) (tr(h, e)) just before executing \( e \). For an ordered history \( (h, <) \) with \( h = (T, \text{so}, \text{wr}) \), we assume that \( < \) is consistent with \( \text{po}, \text{so}, \) and \( \text{wr} \), i.e., \( e_1 < e_2 \) if \((\text{tr}(h, e_1), \text{tr}(h, e_2)) \in \text{(so \cup wr)} \) or \((e_1, e_2) \in \text{po} \).

Initially, the ordered history and the mapping locals are empty.

The function \( \text{EXPLORE} \) starts by calling \( \text{NEXT} \) to obtain an event representing the next database access in some pending transaction of \( P \), or a begin/commit/abort event for starting or ending a transaction. This event is associated to some session \( j \). For example, a typical implementation of \( \text{NEXT} \) would choose one of the pending transactions (in some session \( j \)), execute all local instructions until the next database instruction in that transaction (applying the transition rules \( \text{IF-TRUE}, \text{IF-FALSE}, \) and \( \text{LOCAL} \)) and return the event \( e \) corresponding to that database instruction and the current local state \( \gamma \). \( \text{NEXT} \) may also return \( \bot \) if the program finished.

Otherwise, the event \( e \) is added to the ordered history \( h_\prec \). If \( e \) is a read event, then \( \text{VALID} \) 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 \( \text{wr} \) dependency \( (w, e) \) leads to a history that still satisfies the intended isolation level before outputting the current history and local states (the use of \( \text{VALID} \) will become relevant in Section 6).

### Algorithm 1 \( \text{EXPLORE} \) algorithm

```plaintext
1: function \( \text{EXPLORE}(P, h_\prec, \text{locals}) \)
2: \( j, e, \gamma \leftarrow \text{NEXT}(P, h_\prec, \text{locals}) \)
3: \( \text{locals}' \leftarrow \text{locals}[e \mapsto \gamma] \)
4: if \( e = \bot \) and \( \text{VALID}(h) \) then
5: \( \text{output } h, \text{locals}' \)
6: else if \( \text{type}(e) = \text{read} \) then
7: for all \( t \in \text{VALID} \text{WRIT}E(h, e) \) do
8: \( h'_\prec \leftarrow h_\prec \oplus j \oplus e \oplus \text{wr}(t, e) \)
9: \( \text{EXPLORE}(P, h'_\prec, \text{locals}') \)
10: \( \text{EXPLORE} \text{SWAPS}(P, h'_\prec, \text{locals}') \)
11: else
12: \( h'_\prec \leftarrow h_\prec \oplus j \oplus e \)
13: \( \text{EXPLORE}(P, h'_\prec, \text{locals}') \)
14: \( \text{EXPLORE} \text{SWAPS}(P, h'_\prec, \text{locals}') \)
```

SWAPPING-BASED MODEL CHECKING ALGORITHMS

We define a class of stateless 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 \( \text{EXPLORE} \) listed in Algorithm 1. The function \( \text{EXPLORE} \) receives as input a program \( P \), an ordered history \( h_\prec \), 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 \( e \) (tr(h, e)) just before executing \( e \). For an ordered history \( (h, <) \) with \( h = (T, \text{so}, \text{wr}) \), we assume that \( < \) is consistent with \( \text{po}, \text{so}, \) and \( \text{wr} \), i.e., \( e_1 < e_2 \) if \((\text{tr}(h, e_1), \text{tr}(h, e_2)) \in \text{(so \cup wr)} \) or \((e_1, e_2) \in \text{po} \). Initially, the ordered history and the mapping locals are empty.

The function \( \text{EXPLORE} \) starts by calling \( \text{NEXT} \) to obtain an event representing the next database access in some pending transaction of \( P \), or a begin/commit/abort event for starting or ending a transaction. This event is associated to some session \( j \). For example, a typical implementation of \( \text{NEXT} \) would choose one of the pending transactions (in some session \( j \)), execute all local instructions until the next database instruction in that transaction (applying the transition rules \( \text{IF-TRUE}, \text{IF-FALSE}, \) and \( \text{LOCAL} \)) and return the event \( e \) corresponding to that database instruction and the current local state \( \gamma \). \( \text{NEXT} \) may also return \( \bot \) if the program finished.
in so(j) as the last event in po (i.e., if so(j) = σ; ⟨t, E, po⟩), then the session order so’ of h ⊕ e is defined by so’(k) = so(k) for all k ≠ j and so(j) = σ; ⟨t, E ∪ {e}, po ∪ {(e’, e) : e’ ∈ E}⟩. This is extended to ordered histories: (h, <) ⊕ j e is defined as (h ⊕ j e, < · e) where < · e means that e is added as the last element of <. Also, h ⊕ j (e, begin) is a history where ⟨t, {⟨e, begin⟩}, ∅⟩ with t a fresh id is appended to so(j), and h ⊕ wr(t, e) is defined by adding (t, e) to the write-read of h.

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. 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, writes scheduled later by Next cannot be read by read events executed in the past, although this may be permitted by the 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 would contain a read event r and β a write event w such that re-ordering the two enables r to read from w. Ensuring soundness and 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 Swap(h, α, β, locals) returns pairs (h’, locals’) such that

1. h’ contains at least the events in α and β,
2. h’ without the events in α is a prefix of h, and
3. if a read r in α reads from different writes in h and h’ (the wr relations of h and h’ associate different transactions to r), then r is the last event in its transaction (w.r.t. po).

The first condition makes the re-ordering “meaningful” while the last two conditions ensure that the history h’ is feasible by construction, i.e., it can be obtained using the operational semantics defined in Section 2.3. Feasibility of h’ is ensured by keeping prefixes of transaction logs from h and all their wr dependencies except possibly for read events in α (second condition). In particular, for events in β, it implies that h’ contains all their (po ∪ so ∪ wr) predecessors. Also, the change of a read-from dependency is restricted to the last read in a transaction (third condition) because changing the value returned by a read may disable later events in the same transaction.

A concrete implementation of explore is called:
- I-sound if it outputs only histories in histI(P) for every program P,
- I-complete if it outputs every history in histI(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 another recursive call to EXPLORE.

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

We define a concrete implementation of EXPLORE, denoted as EXPLORE-ce, that is I-sound, I-complete, and strongly optimal for any isolation level I that is prefix-closed and causally-extensible.

---

4 Different wr dependencies for previous reads can be explored in other steps of the algorithm.
The isolation level $I$ is a parameter of EXPLORE-CE. The space complexity of EXPLORE-CE 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 in 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 $I$. Moreover, this invariant is sufficient to guarantee completeness in the sense defined above of exploring all histories of “full” program executions (that end in a final configuration).

Section 5.1 describes the implementations of NEXT and VALIDWrites used to extend a given execution, Section 5.2 describes the functions COMPUTEREORDERINGS and SWAP used to compute re-ordered executions, and Section 5.3 describes the OPTIMALITY restriction on re-ordering. We assume that the function VALID is defined as simply $\text{VALID}(h) ::= \text{true}$ (no filter before outputting). Section 5.4 discusses correctness arguments.

### 5.1 Extending Histories According to An Oracle Order

The function NEXT generates events representing database accesses to extend an execution, according to an arbitrary but fixed order between the transactions in the program called oracle order. We assume that the oracle order, denoted by $<_\text{or}$, is consistent with the order between transactions in the same session of the program. The extension of $<_\text{or}$ to events is defined as expected. For example, assuming that each session has an id, an oracle order can be defined by an order on session ids along with the session order $\text{so}$: transactions from sessions with smaller ids are considered first and the order between transactions in the same session follows $\text{so}$.

NEXT returns a new event of the transaction that is not already completed and that is minimal according to $<_\text{or}$. In more detail, if $j, e, y$ is the output of NEXT($P, h, _<\text{local}$), then either:

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

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.

For example, consider the program in Figure 8a, an oracle order which orders the two transactions in the left session before the transaction in the right session, and the history $h$ in Figure 8b. Since the local state of the pending transaction on the left stores 3 to the local variable $a$ (as a result...
After extending the current history with one more event, explore may be called recursively on other histories obtained by re-ordering events in the current one (and dropping some other events). Re-ordering events must preserve the invariant of producing histories with at most one pending write on the transaction on the left. Assume that the current history reached by the exploration is the one pictured in Figure 10b (the last added event is write(x, 2)). Swapping write(x, 2) with read(x) would result in the history pictured in Figure 10c. To ensure that this swap produces a new history which was not explored in the past, the wrx 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 wry to the only available write(y, ...) event from init. This is pictured in Figure 10d. The next exploration step will extend the history with write(y, 2) (the only extension possible) which however, results of the previous read(x) event) and the Boolean condition in if holds, next will return the event write(y, 1) when called with h.

According to Algorithm 1, 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 ⊕j on ordered histories). If it is a read event, then adding this event may result in multiple histories depending on the chosen wr dependency. For example, in Figure 9, extending the history in Figure 9b with the read(x) event could result in two different histories, pictured in Figure 9c and 9d, depending on the write with whom this read event is associated by wr. However, under CC, the latter history is inconsistent. The function ValidWrites limits the choices to those that preserve consistency with the intended isolation level I, i.e.,

\[
\text{ValidWrites}(h, e) := \{t \in \text{commTrans}(h) \mid h \oplus e \oplus wr(t, e) \text{ satisfies } I\}
\]

where commTrans(h) is the set of committed transactions in h.

### 5.2 Re-Ordering Events in Histories

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

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 10a 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 10b (the last added event is write(x, 2)). Swapping write(x, 2) with read(x) would result in the history pictured in Figure 10c. To ensure that this swap produces a new history which was not explored in the past, the wrx 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 wry to the only available write(y, ...) event from init. This is pictured in Figure 10d. The next exploration step will extend the history with write(y, 2) (the only extension possible) which however, results

![Diagram](attachment:diagram.png)
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 10d which has a pending transaction that is not \((\text{so} \cup \text{wr})^+\)-maximal. Being able to extend such a transaction while maintaining consistency 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 \((\text{so} \cup \text{wr})^+\)-maximal. We enforce this requirement by restricting the explored histories to have at most one pending transaction. This pending transaction will necessarily be \((\text{so} \cup \text{wr})^+\)-maximal.

To enforce histories with at most one pending transaction, the function \textsc{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)\(^5\). Therefore, in such a case, it returns pairs of some transaction log prefix ending in a read \(r\) and the last completed transaction log \(t\), such that the transaction log containing \(r\) and \(t\) are not causally dependent (i.e., related by \((\text{so} \cup \text{wr})^+\)) (the transaction log prefix ending in \(r\) and \(t\) play the role of the subsequences \(\alpha\) and respectively, \(\beta\) in the description of \textsc{ComputeReorderings} from Section 4). To simplify the notation, we will assume that \textsc{ComputeReorderings} returns pairs \((r, t)\).

\textsc{ComputeReorderings}(\(h\)) := \{(r, t) ∈ \mathcal{E} × T | r ∈ \text{reads}(T) ∧ t \text{ writes } \text{var}(r) ∧ \text{tr}(h, r) < t ∧ (\text{tr}(h, r), t) \notin (\text{so} \cup \text{wr})^+ ∧ t \text{ is complete and it includes the last event in } <\}\)

For example, for the program in Figure 11a and history \(h\) in Figure 11b, \textsc{ComputeReorderings}(\(h\)) would return \((r_1, t_4)\) and \((r_2, t_4)\) where \(r_1\) and \(r_2\) are the read(x) events in \(t_1\) and \(t_2\) respectively.

For a pair \((r, t)\), the function \textsc{Swap} produces a new history \(h'\) which contains all the events ordered before \(r\) \((\text{w.r.t. } <)\), the transaction \(t\) and all its \((\text{so} \cup \text{wr})^+\) predecessors, and the event \(r\) reading from \(t\). All the other events are removed. Note that the \text{po} predecessors of \(r\) from the same transaction are ordered before \(r\) by \(<\) and they will be also included in \(h'\). 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.

\textsc{Swap}(\(h\), \(r\), \(t\), locals) := \{(h' = (h \setminus D) ⊕ \text{wr}(t, r), <'), locals'\}, where locals' = locals \downarrow \text{events}(h')\n
\(D = \{e | r < e ∧ (\text{tr}(h, e), t) \notin (\text{so} \cup \text{wr})^+\} ∧ <' = (\text{events}(h')) \setminus \text{events}(\text{tr}(h', r)))\) · \text{tr}(h', r)

\(^5\)Aborted transactions have no visible effect on the state of the database so swapping an aborted transaction cannot produce a new meaningful history.

Fig. 11. Re-ordering events. All so-edges from \text{init} to other transactions are omitted for legibility. The history order \(<\) is represented by the top to bottom order in each figure. Events in gray are deleted from the history.
5.3 Ensuring Optimality

Simply extending histories according to NEXT and making recursive calls on re-ordered histories whenever they are I-consistent guarantees soundness and completeness, but it does not guarantee optimality. Intuitively, the source of redundancy is related to the fact that applying SWAP on different histories may give the same result.

As a first example, consider the program in Figure 12a with 2 transactions that only read some variable x and 2 transactions that only write to x, each transaction in a different session. Assume that EXPLORE reaches the ordered history in Figure 12b and NEXT is about to return the second reading transaction. EXPLORE will be called recursively on the two histories in Figure 12c and Figure 12d that differ in the write that this last read is reading from (the initial write or the first write transaction). On both branches of the recursion, NEXT will extend the history with the last write transaction written in blue bold font. For both histories, swapping this last write with the first read on x will result in the history in Figure 12e (cf. the definition of COMPUTEREORDERINGS and SWAP). Thus, 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 during the SWAP computation. The histories in Figure 12c and Figure 12d differ in the wr dependency involving the last read, but this difference was discarded during the SWAP computation. To avoid this behavior, SWAP is enabled only on histories where the discarded wr dependencies relate to some “fixed” set of writes, i.e., latest\(^6\) writes w.r.t. < that guarantee consistency by causal extensibility (see the definition of readLatest\(_I\)(\_, \_) below). By causal extensibility, a read r can always read from a write which already belongs to its “causal past”, i.e., predecessors in \((\text{so} \cup \text{wr})^*\) excluding the wr dependency for r. For every discarded wr dependency, it is required that the read reads from the latest such write w.r.t. <. In this example, re-ordering is enabled only when the second

\(^6\)We use latest writes because they are uniquely defined. In principle, other ways of identifying some unique set of writes could be used.
read(x) reads from the initial write; write(x, 2) does not belong to its “causal past” (when the wr dependency of the read itself is excluded).

The restriction above is not sufficient, because the two histories for which SWAP gives the same result may not be generated during the same recursive call (for different wr choices when adding a read). For example, consider the program in Figure 13a that has four sessions each containing a single transaction. EXPLORE may compute the history h pictured in Figure 13b. Before adding transaction t4, EXPLORE can re-order t3 and t2 and then extend with t4 and arrive at the history h1 in Figure 13c. Also, after adding t4, it can re-order t1 and t4 and arrive at the history h2 in Figure 13d. However, swapping the same t1 and t4 in h1 leads to the same history h2, thereby, having two recursive branches that end up with the same input and violate optimality. Swapping t1 and t4 in h1 should not be enabled because the read(y) to be removed by SWAP has been swapped in the past. Removing it makes it possible that this recursive branch explores that wr choice for read(y) again.

The Optimality condition restricting re-orderings requires that the re-ordered history be I-consistent and that every read deleted by SWAP or the re-ordered read r (whose wr dependency is modified) reads from a latest valid write, cf. the example in Figure 12, and it is not already swapped, cf. the example in Figure 13 (the set D is defined as in SWAP):

\[
\text{Optimality}(h_\prec, r, t, \text{locals}) := \text{the history returned by SWAP}(h_\prec, r, t, \text{locals}) \text{ satisfies I} \\
\quad \land \forall r' \in \text{reads}(h) \cap (D \cup \{r\}). \neg \text{SWAPPED}(h_\prec, r') \land \text{readLatest}_I(h_\prec, r', t)
\]

A read r reads from a causally latest valid transaction, denoted as readLatest\(_I(h_\prec, r)\), if reading from any other later transaction \(t'\) w.r.t. \(<\) which is in the “causal past” of \(tr(h_\prec, r)\) violates the isolation level I. Formally, assuming that \(t_r\) is the transaction such that \((t_r, r) \in \text{wr}\) in \(h\),

\[
\text{readLatest}_I(h_\prec, r, t) := t_r = \max \left\{ t' \text{ writes } \var(r) \land (t', tr(h_\prec, r)) \in (\text{so} \cup \text{wr})^+ \text{ in } h' \right\} \\
\quad \land \ h' \ominus r \ominus \text{wr}(t', r) \models I
\]

where \(h' = h \setminus \{e \mid r \leq e \land (tr(h, e), t) \notin (\text{so} \cup \text{wr})^+\}\).

We say that a read r is swapped in \(h_\prec\) when (1) \(r\) reads from a transaction \(t\) that is a successor in the oracle order \(\prec_{\text{or}}\) (the transaction was added by NEXT after the read), which is now a predecessor\(^7\) in the history order \(<\), (2) there is no transaction \(t'\) that is before \(r\) in both \(<_{\text{or}}\) and \(<\), and which is a \((\text{so} \cup \text{wr})^+\) successor of \(t\), and (3) \(r\) is the first read in its transaction to read from \(t\). Formally, assuming that \(t\) is the transaction such that \((t, r) \in \text{wr}\),

\[
\text{SWAPPED}(h_\prec, r) := t < r \land t >_{\text{or}} r \land \forall t' \in h. t' <_{\text{or}} tr(h, r) \Rightarrow (r < t' \lor (t, t') \notin (\text{so} \cup \text{wr})^+) \\
\quad \land \forall r' \in \text{reads}(h). (t, r') \in \text{wr} \Rightarrow (r', r) \notin \text{po}
\]

\(^7\)The EXPLORE maintains the invariant that every read follows the transaction it reads from in the history order \(<\).
Condition (1) states a quite straightforward fact about swaps: r could not have been involved in a swap if it reads from a predecessor in the oracle order which means that it was added by Next after the transaction it reads from. Conditions (2) and (3) are used to exclude spurious classifications as swapped reads. Concerning condition (2), suppose that in a history h we swap a transaction t with respect a (previous) read event r. Later on, the algorithm may add a read r’ reading also from t. Condition (2) forbids r’ to be declared as swapped. Indeed, taking tr(h, r) as an instantiation of t’, tr(h, r) is before r’ in both <or and < and it reads from the same transaction as r’, thereby, being a (so ∪ wr)+ successor of the transaction read by r’. Condition (3) forbids that, after swapping r and t in h, later read events from the same transaction as r can be considered as swapped.

Showing that I-completeness holds despite discarding re-orderings is quite challenging. Intuitively, it can be shown that if some Swap is not enabled in some history h< for some pair (r, t) although the result would be I-consistent (i.e., OPTIMALITY(h<, r, t, locals) does not hold because some deleted read is swapped or does not read from a causally latest transaction), then the algorithm explores another history h’ which coincides with h except for those deleted reads who are now reading from causally latest transactions. Then, h’ would satisfy OPTIMALITY(h<, r, t, locals), and moreover applying Swap on h’ for the pair (r, t) would lead to the same result as applying Swap on h, thereby, ensuring completeness.

5.4 Correctness

The following theorem states the correctness of the algorithm presented in this section:

Theorem 5.1. For any prefix-closed and causally extensible isolation level I, EXPLORE-CE is I-sound, I-complete, strongly optimal, and polynomial space.

I-soundness is a consequence of the VALIDWRITES and OPTIMALITY definitions which guarantee that all histories given to recursive calls are I-consistent, and of the Swap definition which ensures to only produce feasible histories (which can be obtained using the operational semantics defined in Section 2.3). The fact that this algorithm never engages in fruitless explorations follows easily from causal-extensibility which ensures that any current history can be extended with any event returned by Next. Polynomial space is also quite straightforward since the for all loops in Algorithm 1 have a linear number of iterations: the number of iterations of the loop in EXPLORE, resp., EXPLORESWAPs, is bounded by the number of write, resp., read, events in the current history (which is smaller than the size of the program; recall that we assume bounded programs with no loops as usual in SMC algorithms). On the other hand, the proofs of I-completeness and optimality are quite complex.

I-completeness means that for any given program P, the algorithm outputs every history h in histt(P). The proof of I-completeness defines a sequence of histories produced by the algorithm starting with an empty history and ending in h, for every such history h. It consists of several steps:

1. Define a canonical total order < for every unordered partial history h, such that if the algorithm reaches h<′, for some order <′, then < and <′ coincide. This canonical order is useful in future proof steps as it allows to extend several definitions to arbitrary histories that are not necessarily reachable, such as OPTIMALITY or SWAPPED.

2. Define the notion of or-respectfulness, an invariant satisfied by every (partial) ordered history reached by the algorithm. Briefly, a history is or-respectful if it has only one pending transaction and for every two events e, e′ such that e <or e′, either e < e′ or there is a swapped event e in between.

3. Define a deterministic function PREV which takes as input a partial history (not necessarily reachable), such that if h is reachable, then PREV(h) returns the history computed by the algorithm just before h (i.e., the previous history in the call stack). Prove that if a history h is or-respectful, then PREV(h) is also or-respectful.
We evaluate an implementation of explore-ce. The experiments were performed on an Apple M1 with (JPF) [Visser et al. 2004] model checker for Java concurrent programs. As benchmark, we use bounded-size client programs of a number of database-backed applications drawn from the literature. In this section is whether there exists another implementation of explore-ce by just filtering histories before outputting to be consistent with SI or SER.

The proof of strong optimality relies on arguments employed for explore-ce. Given this negative result, we define an implementation of explore-ce, denoted by explore-ce∗. We define an implementation of explore for I, denoted by explore-ce∗(I0, I), which is exactly explore-ce(I0) except that instead of Valid(h) := true, it uses

\[
\text{Valid}(h) := h \text{ satisfies } I
\]

explore-ce∗(I0, I) enumerates exactly the same histories as explore-ce(I0) except that it outputs only histories consistent with I. The following is a direct consequence of Theorem 5.1.

Corollary 6.2. For any isolation levels I0 and I such that I0 is prefix-closed and causally extensible, and I0 is weaker than I, explore-ce∗(I0, I) is I-sound, I-complete, optimal, and polynomial space.

7 Experimental Evaluation

We evaluate an implementation of explore-ce and explore-ce∗ in the context of the Java Pathfinder (JPF) [Visser et al. 2004] model checker for Java concurrent programs. As benchmark, we use bounded-size client programs of a number of database-backed applications drawn from the literature. The experiments were performed on an Apple M1 with 8 cores and 16 GB of RAM.
7.1 Implementation

We implemented our algorithms as an extension of the DFS2rch class in JPF. For performance reasons, we implemented an iterative version of these algorithms where roughly, inputs to recursive calls are maintained as a collection of histories instead of relying on the call stack. For checking consistency of a history with a given isolation level, we implemented the algorithms proposed by Biswas and Enea [2019]. We plan to make our implementation publicly available.

Our tool takes as input a Java program and isolation levels as parameters. We assume that the program uses a fixed API for interacting with the database, similar to a key-value store interface. This API consists of specific methods for starting/ending a transaction, and reading/writing a global variable. The fixed API is required for being able to maintain the database state separately from the JVM state (the state of the Java program) and update the current history in each database access. This relies on a mechanism for "transferring" values read from the database state to the JVM state.

7.2 Benchmark

We consider a set of benchmarks inspired by real-world applications and evaluate them under different types of client programs and isolation levels.

Shopping Cart [Sivaramakrishnan et al. 2015] allows users to add, get and remove items from their shopping cart and modify the quantities of the items present in the cart.

Twitter [Difallah et al. 2013] allows users to follow other users, publish tweets and get their followers, tweets and tweets published by other followers.

Courseware [Nair et al. 2020] manages the enrollment of students in courses in an institution. It allows to open, close and delete courses, enroll students and get all enrollments. One student can only enroll to a course if it is open and its capacity has not reached a fixed limit.

Wikipedia [Difallah et al. 2013] allows users to get the content of a page (registered or not), add or remove pages to their watching list and update pages.

TPC-C [TPC 2010] models an online shopping application with five types of transactions: reading the stock of a product, creating a new order, getting its status, paying it and delivering it.

SQL tables are modeled using a “set” global variable whose content is the set of ids (primary keys) of the rows present in the table, and a set of global variables, one variable for each row in the table (the name of the variable is the primary key of that row). SQL statements such as INSERT and DELETE statements are modeled as writes on that “set” variable while SQL statements with a WHERE clause (SELECT, JOIN, UPDATE) are compiled to a read of the table’s set variable followed by reads or writes of variables that represent rows in the table (similarly to [Biswas et al. 2021]).

7.3 Experimental Results

We designed three experiments where we compare the performance of a baseline model checking algorithm, explore-ce and explore-ce∗ for different (combinations of) isolation levels, and we explore the scalability of explore-ce when increasing the number of sessions and transactions per session, respectively. For each experiment we report running time, memory consumption, and the number of end states, i.e., histories of complete executions and in the case of explore-ce∗, before applying the Valid filter. As the number of end states for a program on a certain isolation level increases, the running time of our algorithms naturally increases as well.

The first experiment compares the performance of our algorithms for different combinations of isolation levels and a baseline model checking algorithm that performs no partial order reduction. We consider as benchmark five (independent) client programs8 for each application described above.

8For an application that defines a number of transactions, a client program consists of a number of sessions, each session containing a sequence of transactions defined by the application.
Fig. 14. Cactus plots comparing different algorithms in terms of time, memory, and end states. For readability, we use CC to denote EXPLORE-ce under CC, $I_1 + I_2$ stands for EXPLORE-ce*(I1, I2), and true is the trivial isolation level where every history is consistent. Differences between CC, CC + SI and CC + SER are very small and their graphics overlap. Moreover, DFS(CC) denotes a standard DFS traversal of the semantics defined in Section 2.3. These plots exclude benchmarks that timeout (30 mins): 3 benchmarks for CC, ⟨SI, CC⟩ and ⟨SER, CC⟩ and 6, 17, 20 and 20 benchmarks timeout for ⟨RA, CC⟩, ⟨RC, CC⟩, ⟨true, CC⟩ and DFS(CC) respectively.

(25 in total), each program with 3 sessions and 3 transactions per session. Running time, memory consumption, and number of end states are reported in Fig. 14 as cactus plots [Brain et al. 2017].

To justify the benefits of partial order reduction, we implement a baseline model checking algorithm DFS(CC) that performs a standard DFS traversal of the execution tree w.r.t. the formal semantics defined in Section 2.3 for CC (for fairness, we restrict interleavings so that at most one transaction is pending at a time). This baseline algorithm may explore the same history multiple times since it includes no partial order reduction mechanism. In terms of time, DFS(CC) behaves poorly: it timeouts for 20 out of the 25 programs and it is less efficient even when it terminates. We consider a timeout of 30 mins. In comparison the strongly optimal algorithm EXPLORE-ce(CC) (under CC) finishes in in 3′26″ seconds in average (counting timeouts). DFS(CC) is similar to EXPLORE-ce(CC) in terms of memory consumption. The memory consumption of DFS(CC) is 381MB in average, compared to 508MB for EXPLORE-ce(CC) (JPF forces a minimum consumption of 256MB).

To show the benefits of strong optimality, we compare EXPLORE-ce(CC) which is strongly optimal with “plain” optimal algorithms EXPLORE-ce*(I0, CC) for different levels I0. As shown in Figure 14(a), EXPLORE-ce(CC) is more efficient time-wise than every “plain” optimal algorithm, and the difference in performance grows as I0 becomes weaker. In the limit, when I0 is the trivial isolation level true where every history is consistent, EXPLORE-ce*(true, CC) timeouts for 20 out of the 25 programs. The average speedup (average of individual speedups) of EXPLORE-ce(CC) w.r.t. EXPLORE-ce*(RA, CC), EXPLORE-ce*(RC, CC) and EXPLORE-ce*(true, CC) is 3, 18 and 15. respectively (we exclude timeout cases when computing speedups). All algorithms consume around 500MB of memory in average.

For the SI and SER isolation levels that admit no strongly optimal EXPLORE algorithm, we observe that the overhead of EXPLORE-ce*(CC, SI) or EXPLORE-ce*(CC, SER) relative to EXPLORE-ce(CC) is negligible (the corresponding lines in Figure 14 are essentially overlapping). This is due to the fact that the consistency checking algorithms of Biswas and Enea [2019] are polynomial time when the number of sessions is fixed, which makes them fast at least on histories with few sessions.

In our second experiment, we investigate the scalability of EXPLORE when increasing the number of sessions. For each $i \in [1, 5]$, we consider 5 (independent) client programs for TPC-C and 5 for Wikipedia (10 in total) with $i$ sessions, each session containing 3 transactions. We start with 10 programs with 5 sessions, and remove sessions one by one to obtain programs with fewer sessions. We take CC as isolation level. The plot in Figure 15a shows average running time and memory consumption for each number $i \in [1, 5]$ of sessions. As expected, increasing the number of sessions
is a bottleneck running time wise because the number of histories increases significantly. However, memory consumption does not grow with the same trend, cf. the polynomial space bound.

Finally, we evaluate the scalability of explore-ce (CC) when increasing the number of transactions per session. We consider 5 (independent) TPC-C client programs and 5 (independent) Wikipedia programs with 3 sessions and \( i \) transactions per session, for each \( i \in \{1, 5\} \). Figure 15b shows average running time and memory consumption for each number \( i \in \{1, 5\} \) of transactions per session. Increasing the number of transactions per session is a bottleneck for the same reasons.

8 RELATED WORK

Checking Correctness of Database-Backed Applications. One line of work is concerned with the logical formalization of isolation levels [Adya et al. 2000; Berenson et al. 1995; Biswas and Enea 2019; Cerone et al. 2015; X3 1992]. Our work relies on the axiomatic definitions of isolation levels introduced by Biswas and Enea [2019], which have also investigated the problem of checking whether a given history satisfies a certain isolation level. Our SMC algorithms rely on these algorithms to check consistency of a history with a given isolation level.

Another line of work focuses on the problem of finding “anomalies”: behaviors that are not possible under serializability. This is typically done via a static analysis of the application code that builds a static dependency graph that over-approximates the data dependencies in all possible executions of the application [Bernardi and Gotsman 2016; Cerone and Gotsman 2018; Fekete et al. 2005; Gan et al. 2020; Jorwekar et al. 2007; Warszawski and Bailis 2017]. Anomalies with respect to a given isolation level then correspond to a particular class of cycles in this graph. Static dependency graphs turn out to be highly imprecise in representing feasible executions, leading to false positives. Another source of false positives is that an anomaly might not be a bug because the application may already be designed to handle the non-serializable behavior [Brutschy et al. 2018; Gan et al. 2020]. Recent work has tried to address these issues by using more precise logical encodings of the application [Brutschy et al. 2017, 2018], or by using user-guided heuristics [Gan et al. 2020]. Another approach consists of modeling the application logic and the isolation level in first-order logic and relying on SMT solvers to search for anomalies [Kaki et al. 2018; Nagar and Jagannathan 2018; Ozkan 2020], or defining specialized reductions to assertion checking [Beillahi et al. 2019a,b]. Our approach, based on SMC, does not generate false positives because we systematically enumerate only valid executions of a program which allows to check for user-defined assertions.

Several works have looked at the problem of reasoning about the correctness of applications executing under weak isolation and introducing additional synchronization when necessary [Balegas et al. 2015; Gotsman et al. 2016; Li et al. 2014; Nair et al. 2020]. These are based on static analysis or logical proof arguments. The issue of repairing applications is orthogonal to our work.

MonkeyDB [Biswas et al. 2021] is a mock storage system for testing storage-backed applications. While being able to scale to larger code, it has the inherent incompleteness of testing. As opposed to
MonkeyDB, our algorithms perform a systematic and complete exploration of executions and can establish correctness at least in some bounded context, and they avoid redundancy, enumerating equivalent executions multiple times. Such guarantees are beyond the scope of MonkeyDB.

**Dynamic Partial Order Reduction.** Abdulla et al. [2017b] introduced the concept of source sets which provided the first strongly optimal DPOR algorithm for Mazurkiewicz trace equivalence. Other works study DPOR techniques for coarser equivalence relations, e.g., [Abdulla et al. 2019; Agarwal et al. 2021; Aronis et al. 2018; Chalupa et al. 2018; Chatterjee et al. 2019]. In all cases, the space complexity is exponential when strong optimality is ensured.

Other works focus on extending DPOR to weak memory models either by targeting a specific memory model [Abdulla et al. 2017a, 2016, 2018; Norris and Demsky 2013] or by being parametric with respect to an axiomatically-defined memory model [Kokologiannakis et al. 2022, 2019; Kokologiannakis and Vafeiadis 2020]. Some of these works can deal with the coarser reads-from equivalence, e.g., [Abdulla et al. 2018; Kokologiannakis et al. 2022, 2019; Kokologiannakis and Vafeiadis 2020]. Our algorithms build on the work of Kokologiannakis et al. [2022] which for the first time, proposes a DPOR algorithm which is both strongly optimal and polynomial space. The definitions of database isolation levels are quite different with respect to weak memory models, which makes these previous works not extensible in a direct manner. These definitions include a semantics for transactions which are collections of reads and writes, and this poses new difficult challenges. For instance, reasoning about the completeness and the (strong) optimality of existing DPOR algorithms for shared-memory is agnostic to the scheduler (Next function) while the strong optimality of our explore-ce algorithm relies on the scheduler keeping at most one transaction pending at a time. In addition, unlike TruSt, explore-ce ensures that no swapped events can be swapped again and that the history order $<$ is an extension of so $\cup$ wr. This makes our completeness and optimality proofs radically different. Moreover, even for transactional programs with one access per transaction, where SER and SC are equivalent, TruSt under SC and explore-ce$^*$ $(I_0, \text{SER})$ do not coincide, for any $I_0 \in \{\text{RC}, \text{RA}, \text{CC}\}$. In this case, TruSt enumerates only SC-consistent histories at the cost of solving an NP-complete problem at each step while the explore-ce$^*$ step cost is polynomial time at the price of not being strongly-optimal. Furthermore, we identify isolation levels (SI and SER) for which it is impossible to ensure both strong optimality and polynomial space bounds with a swapping-based algorithm, a type of question that has not been investigated in previous work.

9 CONCLUSIONS

We presented efficient SMC algorithms based on DPOR for transactional programs running under standard isolation levels. These algorithms are instances of a generic schema, called swapping-based algorithms, which is parametrized by an isolation level. Our algorithms are sound and complete, and polynomial space. Additionally, we identified a class of isolation levels, including RC, RA, and CC, for which our algorithms are strongly optimal, and we showed that swapping-based algorithms cannot be strongly optimal for stronger levels SI and SER (but just optimal). For the isolation levels we considered, there is an intriguing coincidence between the existence of a strongly optimal swapping-based algorithm and the complexity of checking if a given history is consistent with that level. Indeed, checking consistency is polynomial time for RC, RA, and CC, and NP-complete for SI and SER. Investigating further the relationship between strong optimality and polynomial-time consistency checks is an interesting direction for future work.

ACKNOWLEDGEMENTS

We thank anonymous reviewers for their feedback, and Ayal Zaks for shepherding our paper. This work was partially supported by the project AdeCoDS of the French National Research Agency.
REFERENCES


A AXIOMATIC LEVELS: READ COMMITTED AND READ ATOMIC.

Fig. A.1. Axioms defining isolation 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., $rel_1 \circ rel_2 = \{(a, b) | \exists c. (a, c) \in rel_1 \land (c, b) \in rel_2\}$.

The axioms defined above in Figure A.1 define the homonymous isolation levels Read Atomic (also called Repeatable Read in the literature) and Read Committed.
B RULES OF THE OPERATIONAL SEMANTICS (SECTION 2.3).

\[
\text{spawn}
\begin{align*}
\text{fresh} & \quad \text{fresh} & \quad P(j) = \text{begin}; \text{Body}; \text{commit}; S \\
\h, \forall, \overline{B}, P \Rightarrow & \ h \oplus_j (t, (\langle e, \text{begin} \rangle, \emptyset), \forall[j \mapsto \emptyset], \overline{B}[j \mapsto \text{Body}; \text{commit}], P[j \mapsto S]
\end{align*}
\]

\[
\text{if-true}
\begin{align*}
\psi(x) & \mapsto \forall(j)(x) : x \in \overline{x} \text{ true} & \quad \overline{B}(j) = \text{if}(\psi(x))\{\text{Instr}; B\} \\
\h, \forall, \overline{B}, P \Rightarrow & \ h, \forall, \overline{B}[j \mapsto \text{Instr}; B], P
\end{align*}
\]

\[
\text{if-false}
\begin{align*}
\psi(x) & \mapsto \forall(j)(x) : x \in \overline{x} \text{ false} & \quad \overline{B}(j) = \text{if}(\psi(x))\{\text{Instr}; B\} \\
\h, \forall, \overline{B}, P \Rightarrow & \ h, \forall, \overline{B}[j \mapsto B], P
\end{align*}
\]

\[
\text{local}
\begin{align*}
v & = \forall(j)(e) & \quad \overline{B}(j) = a := e; B \\
\h, \forall, \overline{B}, P \Rightarrow & \ h, \forall[j \mapsto e, \langle e, \text{write}(x, v) \rangle], \forall, \overline{B}[j \mapsto B], P
\end{align*}
\]

\[
\text{write}
\begin{align*}
v & = \forall(j)(x) & \quad \text{fresh} & \quad \overline{B}(j) = \text{write}(x, a); B \\
\h, \forall, \overline{B}, P \Rightarrow & \ h \oplus_j (e, \langle e, \text{write}(x, v) \rangle) \text{ satisfies } I \\
\h, \forall, \overline{B}, P \Rightarrow & \ h \oplus_j (e, \langle e, \text{write}(x, v) \rangle), \forall[j \mapsto B], P
\end{align*}
\]

\[
\text{read-local}
\begin{align*}
\text{writes}(\text{last}(h, j)) \text{ contains a write}(x, v) \text{ event} & \quad \text{fresh} & \quad \overline{B}(j) = a := \text{read}(x); B \\
\h, \forall, \overline{B}, P \Rightarrow & \ h \oplus_j (e, \langle e, \text{read}(x) \rangle), \forall[j \mapsto B], P
\end{align*}
\]

\[
\text{read-extern}
\begin{align*}
\text{writes}(\text{last}(h, j)) \text{ does not contain a write}(x, v) \text{ event} & \quad \text{fresh} & \quad \overline{B}(j) = a := \text{read}(x); B \\
h = (T, \text{so}, \text{wr}) & \quad t = \text{last}(h, j) & \quad \text{write}(x, v) \in \text{writes}(t') \text{ with } t' \in \text{commTrans}(h) \text{ and } t' \neq t' \\
h' = (h \oplus_j (e, \text{read}(x))) \oplus \text{wr}(t', e) & \quad h' \text{ satisfies } I \\
\h, \forall, \overline{B}, P \Rightarrow & \ h', \forall[j \mapsto v], \forall, \overline{B}[j \mapsto B], P
\end{align*}
\]

\[
\text{commit}
\begin{align*}
\text{fresh} & \quad \overline{B}(j) = \text{commit} \\
\h, \forall, \overline{B}, P \Rightarrow & \ h \oplus_j (e, \langle e, \text{commit} \rangle), \forall[j \mapsto e], P
\end{align*}
\]

\[
\text{abort}
\begin{align*}
\text{fresh} & \quad \overline{B}(j) = \text{abort}; B \\
\h, \forall, \overline{B}, P \Rightarrow & \ h \oplus_j (e, \langle e, \text{abort} \rangle), \forall[j \mapsto \epsilon], P
\end{align*}
\]

Fig. B.1. An operational semantics for transactional programs. Above, last(h, j) denotes the last transaction log in the session order so(j) of h, and commTrans(h) denotes the set of transaction logs in h that are committed.

Figure B.1 uses the following notation. Let h be a history that contains a representation of so as above. We use h ⊕j ⟨t, E, po⟩ to denote a history where ⟨t, E, po⟩ is appended to so(j). Also, for an event e, h ⊕j e is the history obtained from h by adding e to the last transaction log in so(j) and as a last event in the program order of this log (i.e., if so(j) = σ; ⟨t, E, po⟩, then the session order so’ of h ⊕j e is defined by so’(k) = so(k) for all k ̸= j and so(j) = σ; ⟨t, E \cup \{e\}, po \cup \{(e’; e’ \in E)⟩)). Finally, for a history h = ⟨T, so, wr⟩, h ⊕ wr(t, e) is the history obtained from h by adding ⟨t, e⟩ to the read-write relation.

SPAWN starts a new transaction in a session j provided that this session has no live transaction (B(j) = ϵ). 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. COMMIT confirms the end of a transaction making its writes visible while ABORT ends the transaction’s execution immediately.
C PROOF OF THEOREM 3.4

Theorem 3.4. Causal Consistency, Read Atomic, and Read Committed are causally-extensible.

Proof. Let $I$ be an isolation level in $\{\text{CC, RA, RC}\}$. We show that any commit order $co$ justifying that a history $h$ is $I$-consistent can also be used to justify that a causal extension $h'$ of a $(\text{so} \cup \text{wr})^+$-maximal pending transaction $t$ in $h$ with an event $e$ is $I$-consistent as well. We consider a causal extension $h'$ where if $e$ is a read event, then it reads from the last transaction $t_\omega$ in $co$ such that $t_\omega$ writes $\text{var}(e)$ and $(t_\omega, t) \in (\text{so} \cup \text{wr})^+$. Assume by contradiction that this is not the case. Let $\phi_{\text{CC}}(h', t', e') = t' (\text{so} \cup \text{wr})^+ \text{tr}(h', e')$, $\phi_{\text{RA}}(h', t', e') = t' (\text{so} \cup \text{wr}) \text{tr}(h', e')$ and $\phi_{\text{RC}}(h', t', e') = t' (\text{wr} \circ \text{po}) e'$ be sub-formulas of the axioms defining the corresponding isolation level. Then, $h'$ contains transactions $t_1, t_2, t_3$ such that $t_2$ writes some variable $x$, $t_3$ contains some read event $e'$, $(t_1, e') \in \text{wr}_x$ and $\phi_I(h', t_1, e')$ but $(t_1, t_2) \in co$. The assumption concerning $co$ implies that the extended transaction $t$ is one of $t_1, t_2, t_3$ (otherwise, $co$ would not be a “valid” commit order for $h$). Since $t$ is $(\text{so} \cup \text{wr})^+$-maximal in $h$, we have that $t \notin \{t_1, t_2\}$. If $e$ is not a read event, or if $e$ is a read event different from $e'$, then $t \neq t_3$, as $t_1, t_2$ and $t_3$ would satisfy the same constraints in $h$, which is impossible by the hypothesis. Otherwise, if $e = e'$, then this contradicts the choice we made for the transaction $t_\omega$ that $e$ reads from. Since $(t_1, t_2) \in co$ and $t_2$ writes $\text{var}(e)$, it means that $t_\omega = t_1$ is not maximal w.r.t. $co$ among transactions that write $\text{var}(e)$ and precede $t$ in $(\text{so} \cup \text{wr})^+$. Both cases lead to a contradiction, which implies that $h'$ is $I$-consistent, and therefore the theorem holds. □
D PROOF OF THEOREM 6.1

begin;  begin;
a = read(x);  b = read(y);
write(z,1);  write(z,2);
write(y,1);  write(x,2);
...  ...
commit  commit

(a) Program (2 sessions).

read(x)  read(y)
write(z,1)  write(z,2)
write(y,1)  write(x,2)

(b) History h.

read(x)  read(y)
write(z,1)  write(z,2)
write(y,1)  write(x,2)

(c) Two histories. The top history is called h_1.

read(x)  read(y)
write(z,1)  write(z,2)
write(y,1)  write(x,2)

(d) History h_2.

read(x)  read(y)
write(z,1)  write(z,2)
write(y,1)  write(x,2)

(e) History h_3.

read(x)  read(y)
write(z,1)  write(z,2)
write(y,1)  write(x,2)

(f) History h_11.

read(x)  read(y)
write(z,1)  write(z,2)
write(y,1)  write(x,2)

(g) History h_21.

read(x)  read(y)
write(z,1)  write(z,2)
write(y,1)  write(x,2)

(h) History h_31.

read(x)  read(y)
write(z,1)  write(z,2)
write(y,1)  write(x,2)

(i) History h_32.

read(x)  read(y)
write(z,1)  write(z,2)
write(y,1)  write(x,2)

(j) History ĥ.

Fig. D.1. A program and some partial histories. Events in grey are not yet added to the history. For h_3, h_31 and h_32, the number of events that follow write(y, 1) and write(x, 2) is not important (we use black . . . to signify that).

∅ → h_1 → h_2 → h_3

h_11 → h_21 → h_31 → h_32

Fig. D.2. Summary of all possible execution paths from EXPLORE. Black arrows represent alternative explored options depending on NEXT while dashed arrows are mandatory visited histories from such state.
Theorem 6.1. If $I$ is Snapshot Isolation or Serializability, there exists no explore algorithm that is $I$-sound, $I$-complete, and strongly optimal.

Proof. We consider the program in Figure D.1a, 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 three 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 D.1b 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). A diagram of the proof can be seen in Figure D.2.

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 two histories in Figure D.1c, depending on which of the two reads is returned first by Next. We will continue our discussion with the history $h_1$ on the top of Figure D.1c. The other case is similar (symmetric).

From $h_1$, depending on order defined by Next between write $(z, 1)$ and read $(y)$, explore can be called recursively either on $h_{11}$ in Figure D.1f or on $h_2$ in Figure D.1d. Analogously, from $h_2$ two alternatives arise depending on the order defined by Next between read $(y)$ and the rest of events in the left transaction: exploring $h_{21}$ in Figure D.1g if read $(y)$ is added before write $(y, 1)$ or $h_3$ in Figure D.1e otherwise. Thus, from $h_3$ two alternatives arise when added read $(y)$ depending on where it reads from: $h_{31}$ in Figure D.1h if it reads from init and $h_{32}$ in Figure D.1j if it reads from the left transaction.

However, from histories $h_{11}, h_{21}$ or $h_{31}$ explore will necessarily be called recursively on a history $h$ like in Figure D.1b which does not satisfy $I$, thereby violating strong optimality: explore always explore branches that enlarge the current history. Thus, any explore implementation that is strong optimal should only explore $h_{32}$. In such case, by the restrictions on the Swap function (defined in Section 4), any extension of $h_{32}$ does not allow to explore the history $\hat{h}$ in Figure D.1e where read $(x)$ reads from write $(x, 2)$: any outcome of a re-ordering between two contiguous subsequences $\alpha$ and $\beta$ must be prefix of such extension when the events in $\alpha$ are taken out. In particular, for any extension $h'$ of $h_{32}$ and pair of contiguous sequences $\alpha, \beta$ such that $h' \setminus \alpha$ is a prefix of $h'$, if an event from the second transaction belongs to $\beta$, read $(y)$ must also be in $\beta$. Therefore, write $(x, 2)$ must be in $\beta$ as it is $wr^{-1}(\text{read}(y))$. Hence, read $(x)$ must also be in $\beta$. Analogously, if read $(x)$ belongs to $\beta$, init belongs to it. Altogether, if $\beta$ contains any element, then $\alpha$ must be empty; so no swaps can be produced from $h_{32}$. To conclude, in this case explore violates $I$-completeness.

$\Box$
E  PROOF OF THEOREM 5.1

THEOREM 5.1. For any prefix-closed and causally extensible isolation level $I$, EXPLORE-CE is $I$-sound, $I$-complete, strongly optimal, and polynomial space.

As explained in Section 5.4, $I$-soundness, the polynomial space bound, and the part of strong completeness that refers to not engaging in fruitless explorations follow directly from definitions. In the following, we focus on $I$-completeness and then optimality. For the sake of the proof’s readability, we will omit all local states of the algorithm’s definition during the proof. Therefore, we consider programs where we can describe all their events.

E.1 Completeness

By definition, EXPLORE-CE is $I$-complete if for any given program $P$, it outputs every history in $\text{hist}_I(P)$. Let $h \in \text{hist}_I(P)$. Our objective is to produce a computable path of ordered histories that lead to $h$ (i.e., a (finite) ordered collection of ordered histories such that $h_0 = \emptyset$ and for every $n$, if $e = \text{next}(h_n)$, either $h_{n+1} = h_n \lor e, h_{n+1} = h_n \lor \text{wr}(e, t)$ for some $t \in h_n$ or $h_{n+1} = \text{swap}(h_n, r, t)$ for some $r, t \in h_n$).

However, the algorithm EXPLORE-CE works with ordered histories. Therefore, we first have to furnish $h$ with a total order called canonical order that, if $h$ were reachable, it would coincide with its history order. Secondly, we describe a function $\text{prev}$ defined over the set of all partial histories that, if $h$ is reachable, $\text{prev}(h)$ returns the previous history of $h$ computed by EXPLORE-CE. Then, we prove that there exists a finite collection of histories $H = \{h_i\}_{i=0}^n$ such that $h_n = h, h_0 = \emptyset$ and $h_i = \text{prev}(h_{i+1})$. As it ends in the initial state, we can therefore prove that this collection conforms an actual computable path; which allow us to conclude that $h$ is reachable. Nevertheless, for proving both the equivalence between history order and canonical order and the soundness of function $\text{prev}$ we will define the notion of or-respectfulness, an invariant satisfied by every reachable history based on the events’ relative positions in the oracle order.

E.1.1 Canonical order.

As mentioned, we need to formally define a total order for every history that coincide on reachable histories with the history order. For achieving it, we analyze how the algorithm orders transaction logs in a history. In particular, we observe that if two transactions $t, t'$ have a $(\text{so} \cup \text{wr})^*$ dependency, the history order in the algorithm orders them analogously. But if they are $(\text{so} \cup \text{wr})^*$-incomparable, the algorithm prioritizes the one that is read by a smaller read event according or. Combining both arguments recursively we obtain a canonical order for a history, which is formally defined with the function presented below.

The function $\text{canonicalOrder}$ produces a relation between transactions in a history, denoted $\preceq^h$. In algorithm 3’s description, we denote $\perp$ to represent the end of the program, which always exists, and that is so-related with every single transaction.

Firstly, we prove our canonical order is well defined for every pair of transactions.

**Lemma E.1.** For every history $h$, event $e$ and transaction $t$, $\text{dep}(h, t, \min_{\perp} \text{dep}(h, t, e)) \subseteq \text{dep}(h, t, e)$. Moreover, if $\text{dep}(h, t, e) \neq t$, the inclusion is strict.

**Proof.** Let $r' = \min_{\perp} \text{dep}(h, t, e)$. If $\text{dep}(h, t, r') = t$ the lemma is trivially proved, so let’s suppose there exists $r \in \text{dep}(h, t, r') \setminus t$. Then, $\exists t' \text{ s.t. } t \left[ (\text{so} \cup \text{wr})^* t' \land t' [\text{wr}] r \land t' [\text{tr}(h, r)] [\text{so} \cup \text{wr}]^* t' \right]$. Moreover, $t \in \text{so} \cup \text{wr}$ and $t' \in \text{tr}(h, r')$. In other words, $r \in \text{dep}(h, t, e)$. The moreover comes trivially as $r' \notin \text{dep}(h, t, r')$. \hfill $\square$

Algorithm 3 Canonical order

1: procedure canonicalOrder\( (h, t, t') \)
2: \( \text{return } t [\text{so} \cup \text{wr}]^* t' \lor \)
3: \( \neg(t' [\text{so} \cup \text{wr}]^* t) \land \text{minimalDependency}(h, t, t', \bot) \)

4: procedure minimalDependency\( (h, t, t', e) \)
5: let \( a = \min_{\text{or}} \text{dep}(h, t, e); a' = \min_{\text{or}} \text{dep}(h, t', e) \)
6: if \( a \neq a' \) then
7: \( \text{return } a <_{\text{or}} a' \)
8: else
9: \( \text{return } \text{minimalDependency}(h, t, t', a) \)
10: procedure \( \text{dep}(h, t, e) \)
11: \( \text{return } \{ r \mid \exists t' \text{ s.t. } t [\text{so} \cup \text{wr}]^* t' \land t' [\text{wr}] r \land tr(h, r) [\text{so} \cup \text{wr}]^* tr(h, e) \} \cup t \)

Lemma E.2. For every pair of distinct transactions \( t, t' \), \( \text{minimalDependency}(h, t, t', \bot) \) always halts.

Proof. Let’s suppose by contrapositive that \( \text{minimalDependency}(h, t, t', \bot) \) does not halt. Therefore, there would exist an infinite chain of events \( e_n, n \in \mathbb{N} \) such that \( e_0 = \bot, e_{n+1} = \min_{\text{or}} \text{dep}(h, t, e_n) = \min_{\text{or}} \text{dep}(h, t', e_n) \). Firstly, as \( h \) is finite, so are both \( \text{dep}(h, t, e_n) \) and \( \text{dep}(h, t', e_n) \). Moreover, if \( e_n \notin t', \text{dep}(h, t, e_n+1) \subseteq \text{dep}(h, t, e_n) \) (and analogously for \( t' \)). Therefore, there exist some indexes \( n_0, m_0 \) such that \( e_{n_0} \in t \) and \( e_{m_0} \in t' \). Let \( k = \max\{n_0, m_0\} \). Because \( t \), but if \( e_n \in t, t = \text{dep}(h, t, e_n) \) and \( e_{n+1} = e_n \), so \( e_k = e_{n_0} \) and \( e_k = e_{m_0} \). Therefore \( e_k \in t \cap t' \); so \( t = t' \) as transaction logs do not share events; which contradict the assumptions. □

Corollary E.3. The relation \( \leq^h \) is well defined for every pair of transactions.

Proof. As by lemma E.2, we know that \( \text{minimalDependency}(h, t, t', \bot) \) always halts if \( t \neq t' \); it is clear that \( \text{canonicalOrder}(h, t, t') \) also does it. Therefore, the relation is well defined. □

Now that \( \leq^h \) has been proved a well defined relation between each pair of transactions, let us prove that it is indeed a total order.

Lemma E.4. The relation \( \leq^h \) is a total order.

Proof.

- Strongly connection Let \( t_1, t_2 \) s.t. \( t_1 \nleq^h t_2 \). If \( t_2 [\text{so} \cup \text{wr}]^* t_1 \), then \( t_2 \leq^h t_1 \). Otherwise, as \( \neg(t_1 [\text{so} \cup \text{wr}]^* t_2) \) and \( \text{minimalDependency} \) halts (lemma E.2) either \( \text{minimalDependency}(h, t_1, t_2, \bot) \) or \( \text{minimalDependency}(h, t_2, t_1, \bot) \) holds. But as \( t_1 \nleq^h t_2 \), \( t_2 \leq^h t_1 \).

- Reflexivity: By definition, for every \( t, t \leq^h t \).

- Transitivity: Let \( t_1, t_2, t_3 \) three distinct transactions such that \( t_1 \leq^h t_2 \) and \( t_2 \leq^h t_3 \). Clearly, if \( t_1 [\text{so} \cup \text{wr}]^* t_3 \), \( t_1 \leq h t_3 \). However, if \( t_3 [\text{so} \cup \text{wr}]^* t_1 \), we would find one of the following three scenarios:
  - \( t_1 [\text{so} \cup \text{wr}]^* t_2 \), which is impossible by strong connectivity as that would mean \( t_3 \leq^h t_2 \).
  - \( t_2 [\text{so} \cup \text{wr}]^* t_3 \), which is also impossible by strong connectivity, as \( t_2 \leq^h t_1 \).
  - \( \neg(t_1 [\text{so} \cup \text{wr}]^* t_2) \) and \( \neg(t_2 [\text{so} \cup \text{wr}]^* t_3) \). Then, let us call \( e_0^i = \bot \) and \( e_{n+1}^i = \min_{\text{or}} \text{dep}(h, t_i, e_n^i) \) for \( i \in \{1, 2, 3\} \). Let’s prove by induction that if for every \( k < n e_n^i \notin t_i^i \), then \( e_{n+1}^i = e_n^i = e_n^i \). Clearly this hold for \( n = 0 \) and, assuming it holds for every \( k \leq n - 1 \), as \( t_1 \leq^h t_2, t_2 \leq^h t_3 \),

---

Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo
we know $e_1^1 \leq \text{or} e_2^2 \leq \text{or} e_3^3$ and as $t^3 [\text{so} \cup \text{wr}]^* t^1$, if $e_1^1 \notin t^1$, $e_2^2 \leq \text{or} e_3^3$. In other words, they coincide. However, by lemma E.2, we know minimalDependency($h, t^1, t^3, \perp$) halts, so there exists some minimal $n_0$ such that $e_{n_0}^1 \in t^1$; so $e_{n_0}^2 \in t_1$. That implies $t^2 [\text{so} \cup \text{wr}]^* t_1$; which is impossible as $t_1 \leq t_2$.

We deduce then that either $t_1 [\text{so} \cup \text{wr}]^* t_3$ or $\neg(t_3 [\text{so} \cup \text{wr}]^* t_1)$. In the latter case, let’s take the sequence $e_{n_i}^i, i \in \{1, 2, 3\}$ defined in the last paragraph. Then, as by lemma E.2 minimalDependency($h, t_1, t_3, \perp$) halts, there exists a maximum index $n_0$ such that $e_{n_0}^1 = e_{n_0}^3 = e_{n_0}^3$. Then $e_{n_0+1}^i < \text{or} e_{n_0+1}^j$ or $e_{n_0+1}^i < \text{or} e_{n_0+1}^j$; so $t_1 \leq t_3$.

- Antisymmetric Let $t_1, t_2$ s.t. $t_1 \leq t_2$ and $t_2 \leq t_1$. If $t_1 [\text{so} \cup \text{wr}]^* t_2$, then $t_1 = t_2$. If not, by the symmetric argument, $\neg(t_2 [\text{so} \cup \text{wr}]^* t_1)$. In that situation, by lemma E.2 we know both minimalDependency($h, t_1, t_2, \perp$) and minimalDependency($h, t_1, t_2, \perp$) halt and cannot be satisfied at the same time. This contradicts that both $t_1 \leq t_2$ and $t_2 \leq t_1$ hold; so $t_1 = t_2$.

\[\square\]

E.1.2 Oracle-respectful histories.

The second step in this proof is characterizing all reachable histories with some general invariant that can be generalized to every total history. For doing so, we will show that for reachable histories any history order coincide with its canonical order; so any property based on a history order can be generalized to be based on its canonical order.

\textbf{Definition E.5.} An ordered history ($h, \leq$) is \textit{or-respectful} with respect to $\leq$ if it has at most one pending transaction log and for every pair of events $e \in P$, $e' \in h$ s.t. $e \leq \text{or} e'$, either $e \leq e'$ or $\exists e'' \in h, \text{tr}(h, e'') \leq \text{or} \text{tr}(h, e')$ s.t. $\text{tr}(h, e'') [\text{so} \cup \text{wr}]^* \text{tr}(h, e')$, $e'' \leq e$ and swapped($h, e''$); where if $e \notin h$ we state $e' \leq e$ always hold but $e \leq e'$ never does. We will denote it by $\text{R}^\text{or}(h, \leq)$.

\textbf{Lemma E.6.} Let $p$ a computable path. Every ordered history ($h, \leq_h$) in $p$ is or-respectful with respect to $\leq_h$.

\textbf{Proof.} We will prove this property by induction on the number of histories this path has. The base case, the empty path, trivially holds; so let us prove the inductive case: for every path of at most length $n$ the property holds. Let $p$ a path of length $n + 1$ and $h_\leq$ the last reachable history of this path. As $p \setminus \{h\}$ is a computable path of length $n$, the immediate predecessor of $h$ in $p$, $(h_p, <_{h_p})$ is or-respectful with respect to $<_{h_p}$. Let $h = \text{next}(h_p)$.

Firstly, if $a$ is not a read nor a begin event and $h = h_p \oplus a$, as $\leq_h$ is an extension of $\leq_{h_p}$, $a$ belongs to the only pending transaction and or orders transactions completely, we can deduce that $h$ is or-respectful with respect to $\leq$.

In addition, if $a$ is a begin event and $h = h_p \oplus a$, let $e \in P, e' \in h$ s.t. $e <_{\text{or}} e'$. If $e \in h_p$ or $e' \neq a$, as $\leq_h$ is an extension of $\leq_{h_p}$ and $\text{R}^\text{or}(h_p, \leq_{h_p})$ holds, the condition for satisfying $\text{R}^\text{or}(h, \leq)$ holds with $e$ and $e'$. Moreover, as $a = \min_{\text{or}} P \setminus h_p$, there is no event $e \in P \setminus h_p$ s.t. $e <_{\text{or}} a$; so $\text{R}^\text{or}(h, \leq)$ holds.

Moreover, if $a$ is a read event and $h = h_p \oplus \text{wr}(t, a)$ for some transaction log $t$, let us call $e \in P, e' \in h$ s.t. $e <_{\text{or}} e'$. Once again, if $e \in h$ or $e' \neq a$ the property holds; so let’s suppose $e \in P \setminus h_p$ and $e' = a$. Let $b = \text{begin}(\text{tr}(h, a))$, that also belongs to $h_p$. Firstly, as $e <_{\text{or}} \text{tr}(h, e') = \text{tr}(h, b)$ we know that $e <_{\text{or}} b$. Secondly, as $\text{R}^\text{or}(h_p, \leq_{h_p})$, $e \notin h_p$ and $e <_{\text{or}} b$; there exists $c \in h_p, \text{tr}(h_p, c) \leq_{\text{or}} \text{tr}(h_p, a)$ s.t. $(\text{tr}(h_p, b), \text{tr}(h_p, c)) \in ([\text{so} \cup \text{wr}]^*, c \leq b$ and swapped($h_p, <_{h_p}$));. As tr($h, a$) = tr($h, b$) and swapped($h_p, <_{h_p}$), c implies swapped($h, c$, c) we conclude $\text{R}^\text{or}(h, \leq)$.

But if no previous case is satisfied, it is because $h = \text{swap}((h_p, <_{h_p}), r, t)$ for some $r, t \in h_p$ s.t. optimality($h_p, <_{h_p}$), r, t) holds. Let $e, e'$ two events s.t. $e \leq_{\text{or}} e'$. On one hand, if $e <_{\text{or}} e'$, $\text{R}^\text{or}(h, e)$

holds. On the other hand, if \( e' < e \) and \( e' \leq_{h_p} e \), as \( R_{or}(h_p, \leq_{h_p}) \) holds and no swapped event is deleted by \( \text{OPTIMALITY}(\langle h_p, \leq_{h_p} \rangle, r, t) \)'s definition, the property is also satisfied. Finally, if \( e' < e \) and \( e \leq_{h_p} e' \), \( e \) has to be a deleted event so \( e \in P \setminus h \). As \( r \leq_{h_p} e \), if \( e \leq_{or} a \), as \( e \not\leq a \), there would exist a \( c \in h_p, \text{tr}(h_p, c) \leq_{or} \text{tr}(h, r), \text{tr}(h, r, r, c) \in (\text{sor} \cup \text{wr})^* \) and \( \text{SWAPPED}(h_c, c) \). However, this is impossible as \( \text{tr}(h_c, r) \) has as maximal event \( r \) and the algorithm preserves at most one pending transaction; so \( e \leq_{or} a \). Taking \( e'' = r \) the property is witnessed. \( \Box \)

**Proposition E.7.** For any reachable history \( h, \leq^h \equiv \leq_h \).

**Proof.** For proving this equivalence, we will show that in any computable path and for any ordered history \( (h, \leq_h) \), if \( t \leq_h t' \), then \( t \leq^h t' \), as by lemma E.4 \( \leq^h \) is a total order and therefore they have to coincide. We will prove this by induction on the number of histories a path has. The base case, the empty path, trivially holds; so let us prove the inductive case: for every path of at most length \( n \) the property holds. Let \( p \) a path of length \( n + 1 \) and \( h_{<h} \) the last reachable ordered history of this path. As \( p \setminus \{ h \} \) is a computable path of length \( n \), the immediate predecessor of \( h \) in \( p \), \( \leq^h \equiv \leq_{h_p} \). Let \( e = \text{NEXT}(h_p) \). Firstly, let's note that if \( h \) is an extension of \( h_p \), as \( R_{or}(h_p, \leq_{h_p}) \), the property can only fail while comparing a transaction \( t \) with \( \text{tr}(h,e) \).

- \( h \) extends \( h_p \) and \( e \) is a \texttt{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, \( \text{DEP}(h, \text{tr}(h,e), \perp) = \{ e \} = \min_{or} P \setminus h_p \). By lemma E.6 \( h \) is \texttt{or}-respectful, so for every \( t, \min_{or} \text{DEP}(h, t, \perp) \leq_{or} e \); which implies \( t \leq^h \text{tr}(h,e) \). By lemma E.4, \( \leq^h \) is a total order, so it coincides with \( \leq_h \).

- \( h \) extends \( h_p \) and \( e \) is not a \texttt{begin}: As no transaction depends on \( \text{tr}(h,e) \) and \( \text{tr}(h,e) = \text{last}(h_p) \), if we prove that for every pair of transactions \( \text{MINIMALDEPEN} \text{DENCY}(h_p, t', t'', \perp) = \text{MINIMALDEPEN} \text{DENCY}(h_p, t', t'', \perp) \).

- \( h = \text{swap}(h_p, r, t) \), where \( t = \text{tr}(h,e) \): As \( \text{OPTIMALITY}(h_p, r, t) \) is satisfied and \( h \) is \texttt{or}-respectful, for every event \( e' \) and transaction \( t' \) in \( h \), \( \min_{or} \text{DEP}(h_p, t', e') = \min_{or} \text{DEP}(h, t', e') \), so for every pair of transactions \( \text{MINIMALDEPEN} \text{DENCY}(h_p, t', t'', \perp) = \text{MINIMALDEPEN} \text{DENCY}(h, t', t'', \perp) \). In particular, this implies \( t' \leq_{h_p} t'' \) if and only if \( t' \leq^h t'' \) for every pair \( t', t'' \in h \). Finally, as for every \( t' \in h \), \( t' \leq^h \text{tr}(h,r) \) (because \( \text{tr}(h,r) \) is \( \text{so} \cup \text{wr} \)-maximal); we conclude that \( \leq^h \equiv \leq_h \).

**Proposition E.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. Therefore, all possible history orders collapse to one, the canonical one. 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 E.7 together with lemma E.6 justify enlarging definition E.5 with a general order as for reachable histories, \( R_{or}(h, \leq_h) \) is equivalent to \( R_{or}(h, \leq^h) \). From what follows, we will simply state \( h \) is \texttt{or}-respectful and we will denote it by \( R_{or}(h) \). Moreover, we will assume every history is ordered with the canonical order.

**Corollary E.8.** Let \( h_p \) a reachable history and let \( h \) a immediate successor of \( h_p \) whose last event \( r \) is a \texttt{read}. Then \( h_{<h} = \text{swap}(\langle h_p, \leq_{h_p} \rangle, r, t) \) if and only if \( \text{SWAPPED}(h, r) \) does.

**Proof.** \( \Rightarrow \)
Let’s suppose that $h_c = \text{swapped}(h_p, r, t)$ for some $t$ transaction. As the last event in $h$ is $r$ and by definition of swap function no event reads from $\text{wr}^{-1}(r)$ in $h$ besides $r$, to prove $\text{swapped}(h, r)$ holds we just need to show that $r <_{\text{or}} t$. By lemma E.6, $R^\text{or}(h_p)$ holds. As $r <_{h_p} t$, OPTIMALITY$(h_p, r, t)$ holds and $t$ is $(\text{so} \cup \text{wr})^+$-maximal, we conclude that $r <_{\text{or}} t$

$\Longleftrightarrow$ Let’s suppose that $h = h_p \oplus r \oplus \text{wr}(r, t)$ for some transaction $t$. Let’s suppose that $r <_{\text{or}} t$. As $R^\text{or}(h_p)$, there exists some event $e''$ s.t. $\text{tr}(h_p, e'') \leq \text{tr}(h, r), t [\text{so} \cup \text{wr}]^+ \text{tr}(h, e'')$ and $e'' \leq r$ so $\neg(\text{swapped}(h, r))$.

\[\square\]

**Lemma E.9.** 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: $t' \leq t$. If $(t', t) \in (\text{so} \cup \text{wr})^+$, then for every $e \in t, e' \in t'$ $\exists c \in h$ s.t. $\text{tr}(h, c) \leq_{\text{or}} \text{tr}(h, e), (\text{tr}(h, e'), \text{tr}(h, c)) \in (\text{so} \cup \text{wr})^+$, $\text{swapped}(h, c)$ and $c \leq^h e$; so the property is satisfied. Otherwise, by definition of minimaldependency, there exists $r' \in h$ s.t. $(t', \text{tr}(h, r')) \in (\text{so} \cup \text{wr})^+$ and $\text{tr}(h, r') \leq_{\text{or}} t$. Moreover, by canonicalorder’s definition, $\text{tr}(h, r')$ holds as it is the minimum element according or. To sum up, $R^\text{or}(h)$ holds.

\[\square\]

**E.1.3 Previous of a history.**

As a third and final step in our proof, we define the function $\text{previous}$ that, for every history $h$, if $\text{prev}(h)$ is reachable, then $h$ is also reachable. Moreover, $\text{prev}(h)$ will belong to the same computable path.

**Algorithm 4 prev**

```
1: procedure prev(h)
2:   if $h = \emptyset$ then
3:       return $\emptyset$
4:   a ← last(h)
5:   if $\neg\text{swapped}(h, a)$ then
6:       return $h \setminus a$
7:   else
8:       let $t$ s.t. $(t, r) \in \text{wr}$.
9:       return maxcompletion($h \setminus a, \{ e \mid e \notin (h \setminus a) \land e <_{\text{or}} t \}$)
10: end procedure

11: function maxcompletion(h, D)
12:   if $D \neq \emptyset$ then
13:     e ← $\min_{<_{\text{or}}} D$
14:     if type(e) ≠ read then
15:       return maxcompletion($h \oplus e, D \setminus \{ e \}$)
16:     else
17:       let $t$ s.t. readlatest$_t(h \oplus e \oplus \text{wr}(t, e), e)$ holds
18:       return maxcompletion($h \oplus e \oplus \text{wr}(t, e), D \setminus \{ e \}$)
19:     else
20:       return $h$
```

First, we show that the invariant of our algorithm is preserved via $\text{prev}$.
Lemma E.10. For every or-respectful history $h$, $\text{prev}(h)$ is also or-respectful.

Proof. Let suppose $h \neq \emptyset$, $h_p = \text{prev}(h)$, $a = \text{last}(h)$, $e \in P$ and $e' \in h_p$ s.t. $e \leq_{or} e'$. We explore different cases depending if $e, e'$ belong to $h$ or not. If $e' \in h_p \setminus h$, $\neg(\text{swapped}(h_p, e))$ and $\neg(\text{swapped}(h_p, e'))$ holds. As $\min_{or} \text{dep}(h, \text{tr}(h, e'), \bot) = \text{begin}(\text{tr}(h, e'))$, we obtain that $\min_{or} \text{dep}(h, \text{tr}(h, e')) \leq_{or} e' \leq_{or} \text{begin}(\text{tr}(h, e'))$. Therefore, as $e' \in h_p \in h$, $\neg(\text{tr}(h, e'))$ so $e \leq^h e'$. And if $e' \in h$, either $e \leq^h e'$ or $e' \leq^h e$. In the former case, both are in $h$ and therefore, in $h_p$. As it cannot happen that $e' \in \text{tr}(h, a)$ and $e \leq_{h_p} a$ because swapped($h, a$) and $e \leq_{or} e'$, we conclude that $e \leq^h e'$ ($\leq_{h_p}$, keeps the relative orders between transactions different from $\text{tr}(h, a)$ and by lemma E.6 they coincide). In the latter case, by $\text{Ror}(h)$, there exists $e''$ that witness it. In particular, swapped($h, e''$) holds, so $e'' \in h_p$. $e''$ witness $\text{Ror}(h)$ holds. In the three cases we deduce that $\text{Ror}(h_p)$.

Next, we have to prove that previous is a sound function, i.e. the composition between $\text{explore-ce}$ and $\text{prev}$ give us the identity. For doing so, in the case a history is a swap, we deduce that both histories should contain the same elements and they read the same; so they have to coincide.

Lemma E.11. For every consistent history or-respectful $h$, if $\text{prev}(h)$ is reachable, then $h$ is also reachable.

Proof. Let suppose $h \neq \emptyset$, $h_p = \text{prev}(h)$ and $a = \text{last}(h)$. If $\neg(\text{swapped}(h, a))$, let $h_n = h_p \oplus a \oplus \text{wr}(t, a)$, where $t$ is the transaction s.t. $(t, r) \in \text{wr}$, otherwise. Either way, $h_n$ is always reachable and it coincides with $h$. On the contrary, if swapped($h, a$), a is a read event and it swapped; so let us call $t$ to the transaction s.t. $(t, a) \in \text{wr}$. Firstly, as swapped($h, a$), $a <_{or} t$, and by lemma E.6, $\text{Ror}(h_p)$ holds, so $a <_{h_p} t$ does; which let us conclude $\text{compute-reorderings}(h_p)$ will always return $(a, t)$ as a possible swap pair. In addition, all transactions in $h_p$ are non-pending and $(t, a) \in \text{wr}$, so in particular last($h_p$) is an commit event. If we call $h_s = \text{swap}(h_p, a, t)$, and we prove that $h_p \cup \text{wr}(t, a)$ in both $h_p, h_s$ and $h = h_p \cup \text{wr}(t, a)$; which would allow us to conclude $h$ is reachable from $h_p$.

On one hand, if $e \in h_p \setminus h$, we deduce that $e \notin h$ and $e <_{or} t$. In particular, $\neg(\text{tr}(h, e) \ [\text{so} \cup \text{wr}]^* t)$. Moreover, if $e <_{or} a$, by $\text{Ror}(h)$, either $e \leq^h a$ or $\exists e'' \in h, e'' \leq_{or} e$ s.t. t(a) [so $\cup$ wr]$^*\text{tr}(h, e'')$, $e'' \leq^h e$ and swapped($h, e''$); both impossible situations as $e \notin h$ and $a = \text{last}(h)$; so $a <_{or} e$. 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 $\neg(\text{tr}(h, e) \ [\text{so} \cup \text{wr}]^* t(w))$ and $a <_{or} e <_{or} w$. If $e$ would belong to $h$ then $e \leq^h a$. As $h$ is or-respectful and $a \leq_{or} e$, we deduce there exists a $e'' \in h$ s.t. $\text{tr}(h, e'') \leq_{or} t(a)$, $\text{tr}(h, e) \ [\text{so} \cup \text{wr}]^*\text{tr}(h, e'')$ and swapped($h, e''$). Moreover, as $e'' \in h$, $e'' \in h_p$. By corollary E.8 swapped($h_p, e''$) and optimality($h_p, a, t$) hold, $e'' \in h_s$ and so e does. This result leads to a contradiction, so e $\notin h$; i.e. e $\in h_p \setminus h$.

Corollary E.12. In a consistent or-respectful history $h$ whose previous history is reachable, if $a = \text{last}(h)$, swapped($h, a$) and $t$ is a transaction such that $(t, a) \in \text{wr}$, $h$ coincides with swap($\text{prev}(h), a, t$).

Proof. It comes straight away from the proof of lemma E.11. 

Once proven that prev is sound, let us prove that for every history we can compose prev a finite number of times obtaining the empty history. We are going to prove it by induction on the number of swapped events, so we prove first the recursive composition finishes in finite time and then we conclude our claim.
Lemma E.13. 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 \( \neg(\text{SWAPPED}(h, a)) \), then \( h_p = h' \) and the lemma holds trivially. 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 \( \text{MAXCOMPLETION} \). Let us call \( D = \{ e \mid e \notin h' \lor e <_\text{or} \} \). 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 \( h_{D|D} = h' \), \( D_{|D} = D \) and \( D_k = D_{k+1} \setminus \{ \min_{\leq a} D_k \} \), \( e_k = \min_{\leq a} D_k \) 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 or-respectful 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, \( h_{D|D} \) 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, \( h_k = h_{k+1} \oplus e_{k+1} \), \( R^\text{or}(h_k) \) and \( \{ e \mid \text{SWAPPED}(h_k, e) \} = \{ e \mid \text{SWAPPED}(h', e) \} \); as only read events can be swapped. Otherwise, \( e_{k+1} \) is a read event. By the isolation level’s causal-extensibility there exists a transaction \( f_{k+1} \) that writes the same variable as \( e_{k+1} \), \( (f_{k+1}, tr(h, e_{k+1})) \in (\text{so} \cup \text{wr})^* \) and \( h_{k+1} \oplus e_{k+1} \oplus \text{wr}(f_{k+1}, e_{k+1}) \) is consistent. Moreover, if \( e_{k+1} \) reads from any causal dependent element \( f'_{j}, f'_{j} \) in \( h_{k+1} \), it cannot be swapped: as \( R^\text{or}(h_{k+1}) \) holds, if \( e_{k+1} <_\text{or} f'_{j} \) there must be an event \( c_{k+1} \) s.t. \( tr(h, c_{k+1}) \leq_\text{or} fr(h, e_{k+1}) \) and \( (f', tr(h, c_{k+1})) \in (\text{so} \cup \text{wr})^* \). Hence, \( \{ e \mid \text{SWAPPED}(h_{k+1}, e) \} = \{ e \mid \text{SWAPPED}(h_{k+1} \oplus e_{k+1} \oplus \text{wr}(f', e_{k+1}, e)) \} \) .

Let \( E_{k+1} = \{ t \mid h_{k+1} \oplus e_{k+1} \oplus \text{wr}(t, e_{k+1}) \models I \land \{ e \mid \text{SWAPPED}(h_{k+1}, e) \} = s \{ e \mid \text{SWAPPED}(h_{k+1} \oplus e_{k+1} \oplus \text{wr}(t, e_{k+1}, e)) \} \} \) and let \( t_{k+1} = \max_{\leq_{h_{k+1}}} \{ t \in E_{k+1} \mid (t, tr(h_{k+1}, e_{k+1})) \in (\text{so} \cup \text{wr})^* \} \). This element is well defined as \( f_{k+1} \) belongs to \( E_{k+1} \). Therefore, \( h_k = h_{k+1} \oplus e_{k+1} \oplus \text{wr}(t, e_{k+1}, e_{k+1}) \) is consistent and \( \{ e \mid \text{SWAPPED}(h_k, e) \} = \{ e \mid \text{SWAPPED}(h', e) \} \). Moreover, let’s remark that as \( t_{k+1} \) is the maximum transaction according to \( \leq_{h_{k+1}} \) s.t. is consistent and \( \{ e \mid \text{SWAPPED}(h_k, e) \} = \{ e \mid \text{SWAPPED}(h', e) \} \). In addition, by construction, it also satisfies readLatest\(_i\)\((h_k, e_{k+1}, w_{k+1})\). Finally, \( h_k \) is also or-respectful as \( e_{k+1} \) is not swapped and \( R^\text{or}(h_{k+1}) \) holds.

Thus, after applying induction, 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 E.14. For every consistent or-respectful 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 E.13. 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 \( \xi(h) = h \land |h| < m \) 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 \( h_p \neq h \setminus a \) 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 E.15. 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 E.14 and \( C = \{ \text{PREV}^{k-n}(h) \}_{n=0}^k \) a set of indexed histories. By the collection’s definition and lemma E.10, \( h_0 = \text{PREV}^k(h) = \emptyset \), \( h_k = \text{PREV}^0(h) = h \) 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 E.11, we know that if \( h_n \) is reachable, \( h_{n+1} \) is it too; which proves the inductive step. \( \square \)

**Theorem E.16.** The algorithm explore-ce is complete.

**Proof.** By lemma E.9, any consistent total history is or-respectful. As a consequence of proposition E.15, there exist a sequence of reachable histories which \( h \) belongs to; so in particular, \( h \) is reachable. \( \square \)

### E.2 Optimality

For proving optimality we are going to exploit two properties already studied for completeness: or-respectfulness and the canonical order. Then, as algorithm EXPLORE-CE is sound and complete, we will prove that any computable path leading to a consistent history is the one computed in the completeness’ proof.

**Theorem E.17.** Algorithm explore-ce is strongly optimal.

**Proof.** As the model is causal-extensible, any algorithm optimal is also strongly optimal. Let us prove that for every reachable history there is only a computable path that leads to it from \( \emptyset \). Let’s suppose there exists a history \( h \) that is reached \( p_1, p_2 \) by two computable paths. By lemma E.7, we know that \( \leq_h \equiv \leq^h \). However, \( \leq^h \) is an order that does not depend on the computable path that leads to \( h \); so neither does \( \leq_h \). Therefore, we can assume without loss of generality that \( h \) is a history with minimal value of \( \xi(h) = |\{e \in h \mid \text{SWAPPED}(h, e)\}| \) and in case of tie, that is minimal with respect \( |h| \); values independent of the computable path that leads to \( h \).

We can also assume without loss of generality that the predecessor of \( h \) in \( p_1 \) is \( h_1 = \text{prev}_h \), and \( h_2 \) is the predecessor of \( h \) in \( p_2 \). If we prove \( h_1 \) and \( h_2 \) are identical, \( p_1 \) and \( p_2 \) have to also be identical and therefore, the algorithm would be optimal. Firstly, if \( \text{last}(h) \) is not a swapped read event, by the definition of \text{next} function \( h_2 = h \setminus \text{last}(h) = h_1 \). On the contrary, let’s suppose \( r = \text{last}(h) \) is a swapped event that reads from a transaction \( t \). Because \text{SWAPPED}(h, r) \) holds, from \( h_2 \) to \( h \) it has to have happened a swap between \( r \) and \( w \). But by corollary E.12, \( h = \text{SWAP}(h_1, r, w) \), so \( h_1 \setminus h_1 \setminus h_1 \setminus h_1 \setminus h_1 \). As \( h_1, h_2 \) are both or-respectful, \( e \in h_1 \setminus h_1 \iff e \in h_1 \setminus h_1 \). Finally, as \text{Optimality}(h_i, r, w) \) holds for \( i \in \{1, 2\} \), for every read event \( e \) in \( h_1 \cap h_2 \) there exists a transaction \( t_e \) s.t. \( \text{wr}(e, t_e) \) for both histories. \( \square \)
### F.1 Application Scalability

#### F.1.1 Application Scalability

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>courseware-1</td>
<td>216</td>
<td>216</td>
<td>0:00:22</td>
<td>370</td>
<td>216</td>
<td>0:00:25</td>
<td>370</td>
<td>216</td>
<td>0:00:23</td>
<td>370</td>
<td>216</td>
<td>0:00:23</td>
<td>370</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>courseware-2</td>
<td>46</td>
<td>46</td>
<td>0:00:06</td>
<td>316</td>
<td>46</td>
<td>0:00:06</td>
<td>308</td>
<td>46</td>
<td>0:00:06</td>
<td>314</td>
<td>46</td>
<td>0:00:06</td>
<td>314</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>courseware-3</td>
<td>1270</td>
<td>1270</td>
<td>0:00:12</td>
<td>533</td>
<td>6197</td>
<td>12700</td>
<td>13:59</td>
<td>533</td>
<td>12700</td>
<td>12:37</td>
<td>557</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>courseware-4</td>
<td>69</td>
<td>69</td>
<td>0:00:07</td>
<td>314</td>
<td>69</td>
<td>0:00:08</td>
<td>324</td>
<td>69</td>
<td>0:00:07</td>
<td>370</td>
<td>69</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>courseware-5</td>
<td>388</td>
<td>388</td>
<td>0:00:25</td>
<td>308</td>
<td>388</td>
<td>0:00:27</td>
<td>370</td>
<td>388</td>
<td>0:00:24</td>
<td>370</td>
<td>71</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shoppingCart-1</td>
<td>444</td>
<td>444</td>
<td>0:00:19</td>
<td>308</td>
<td>444</td>
<td>0:00:22</td>
<td>308</td>
<td>444</td>
<td>0:00:19</td>
<td>308</td>
<td>81</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shoppingCart-2</td>
<td>2934</td>
<td>2934</td>
<td>0:00:55</td>
<td>308</td>
<td>811</td>
<td>2954</td>
<td>0:11:14</td>
<td>444</td>
<td>480</td>
<td>2954</td>
<td>0:00:58</td>
<td>308</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shoppingCart-3</td>
<td>1594</td>
<td>1594</td>
<td>0:00:55</td>
<td>308</td>
<td>1077</td>
<td>1594</td>
<td>0:01:05</td>
<td>308</td>
<td>338</td>
<td>1594</td>
<td>0:01:00</td>
<td>308</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shoppingCart-4</td>
<td>58677</td>
<td>58677</td>
<td>TL</td>
<td>444</td>
<td>58677</td>
<td>49589</td>
<td>TL</td>
<td>444</td>
<td>379</td>
<td>60194</td>
<td>TL</td>
<td>383</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### F.2 Session Scalability

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>courseware-1</td>
<td>216</td>
<td>216</td>
<td>0:00:22</td>
<td>370</td>
<td>216</td>
<td>0:00:25</td>
<td>370</td>
<td>216</td>
<td>0:00:23</td>
<td>370</td>
<td>216</td>
<td>0:00:23</td>
<td>370</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>courseware-2</td>
<td>46</td>
<td>46</td>
<td>0:00:06</td>
<td>316</td>
<td>46</td>
<td>0:00:06</td>
<td>308</td>
<td>46</td>
<td>0:00:06</td>
<td>314</td>
<td>46</td>
<td>0:00:06</td>
<td>314</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>courseware-3</td>
<td>1270</td>
<td>1270</td>
<td>0:00:12</td>
<td>533</td>
<td>6197</td>
<td>12700</td>
<td>13:59</td>
<td>533</td>
<td>12700</td>
<td>12:37</td>
<td>557</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>courseware-4</td>
<td>69</td>
<td>69</td>
<td>0:00:07</td>
<td>314</td>
<td>69</td>
<td>0:00:08</td>
<td>324</td>
<td>69</td>
<td>0:00:07</td>
<td>370</td>
<td>69</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>courseware-5</td>
<td>388</td>
<td>388</td>
<td>0:00:25</td>
<td>308</td>
<td>388</td>
<td>0:00:27</td>
<td>370</td>
<td>388</td>
<td>0:00:24</td>
<td>370</td>
<td>71</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shoppingCart-1</td>
<td>444</td>
<td>444</td>
<td>0:00:19</td>
<td>308</td>
<td>444</td>
<td>0:00:22</td>
<td>308</td>
<td>444</td>
<td>0:00:19</td>
<td>308</td>
<td>81</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shoppingCart-2</td>
<td>2934</td>
<td>2934</td>
<td>0:00:55</td>
<td>308</td>
<td>811</td>
<td>2954</td>
<td>0:11:14</td>
<td>444</td>
<td>480</td>
<td>2954</td>
<td>0:00:58</td>
<td>308</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shoppingCart-3</td>
<td>1594</td>
<td>1594</td>
<td>0:00:55</td>
<td>308</td>
<td>1077</td>
<td>1594</td>
<td>0:01:05</td>
<td>308</td>
<td>338</td>
<td>1594</td>
<td>0:01:00</td>
<td>308</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shoppingCart-4</td>
<td>58677</td>
<td>58677</td>
<td>TL</td>
<td>444</td>
<td>58677</td>
<td>49589</td>
<td>TL</td>
<td>444</td>
<td>379</td>
<td>60194</td>
<td>TL</td>
<td>383</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### F.3 Transaction Scalability

<table>
<thead>
<tr>
<th></th>
<th>One transaction</th>
<th>Two transactions</th>
<th>Three transactions</th>
<th>Four transactions</th>
<th>Five transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpcc-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histories</td>
<td>4</td>
<td>303</td>
<td>326</td>
<td>348</td>
<td>370</td>
</tr>
<tr>
<td>Mem.</td>
<td>00:00:03</td>
<td>00:00:11</td>
<td>00:00:10</td>
<td>00:00:09</td>
<td>00:00:07</td>
</tr>
<tr>
<td>Time</td>
<td>256</td>
<td>220</td>
<td>520</td>
<td>1026</td>
<td>1707</td>
</tr>
<tr>
<td>tpcc-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histories</td>
<td>18</td>
<td>5162</td>
<td>5162</td>
<td>5162</td>
<td>5162</td>
</tr>
<tr>
<td>Mem.</td>
<td>00:00:11</td>
<td>00:00:10</td>
<td>00:00:09</td>
<td>00:00:09</td>
<td>00:00:09</td>
</tr>
<tr>
<td>Time</td>
<td>444</td>
<td>1022</td>
<td>1022</td>
<td>1022</td>
<td>1022</td>
</tr>
<tr>
<td>tpcc-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histories</td>
<td>3</td>
<td>6679</td>
<td>6679</td>
<td>6679</td>
<td>6679</td>
</tr>
<tr>
<td>Mem.</td>
<td>00:00:13</td>
<td>00:00:13</td>
<td>00:00:13</td>
<td>00:00:13</td>
<td>00:00:13</td>
</tr>
<tr>
<td>Time</td>
<td>444</td>
<td>1063</td>
<td>1063</td>
<td>1063</td>
<td>1063</td>
</tr>
<tr>
<td>tpcc-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histories</td>
<td>1</td>
<td>717</td>
<td>717</td>
<td>717</td>
<td>717</td>
</tr>
<tr>
<td>Mem.</td>
<td>00:00:03</td>
<td>00:00:11</td>
<td>00:00:10</td>
<td>00:00:09</td>
<td>00:00:07</td>
</tr>
<tr>
<td>Time</td>
<td>256</td>
<td>220</td>
<td>520</td>
<td>1026</td>
<td>1707</td>
</tr>
<tr>
<td>wikipedia-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histories</td>
<td>16</td>
<td>22289</td>
<td>22289</td>
<td>22289</td>
<td>22289</td>
</tr>
<tr>
<td>Mem.</td>
<td>00:00:04</td>
<td>00:00:12</td>
<td>00:00:12</td>
<td>00:00:12</td>
<td>00:00:12</td>
</tr>
<tr>
<td>Time</td>
<td>308</td>
<td>444</td>
<td>444</td>
<td>444</td>
<td>444</td>
</tr>
<tr>
<td>wikipedia-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histories</td>
<td>9</td>
<td>739</td>
<td>739</td>
<td>739</td>
<td>739</td>
</tr>
<tr>
<td>Mem.</td>
<td>00:00:06</td>
<td>00:00:20</td>
<td>00:00:20</td>
<td>00:00:20</td>
<td>00:00:20</td>
</tr>
<tr>
<td>Time</td>
<td>256</td>
<td>533</td>
<td>533</td>
<td>533</td>
<td>533</td>
</tr>
<tr>
<td>wikipedia-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histories</td>
<td>18</td>
<td>26110</td>
<td>26110</td>
<td>26110</td>
<td>26110</td>
</tr>
<tr>
<td>Mem.</td>
<td>00:00:07</td>
<td>00:00:12</td>
<td>00:00:12</td>
<td>00:00:12</td>
<td>00:00:12</td>
</tr>
<tr>
<td>Time</td>
<td>308</td>
<td>640</td>
<td>640</td>
<td>640</td>
<td>640</td>
</tr>
<tr>
<td>wikipedia-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histories</td>
<td>4</td>
<td>3501</td>
<td>3501</td>
<td>3501</td>
<td>3501</td>
</tr>
<tr>
<td>Mem.</td>
<td>00:00:05</td>
<td>00:00:12</td>
<td>00:00:12</td>
<td>00:00:12</td>
<td>00:00:12</td>
</tr>
<tr>
<td>Time</td>
<td>256</td>
<td>444</td>
<td>444</td>
<td>444</td>
<td>444</td>
</tr>
<tr>
<td>wikipedia-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histories</td>
<td>2</td>
<td>754</td>
<td>754</td>
<td>754</td>
<td>754</td>
</tr>
<tr>
<td>Mem.</td>
<td>00:00:05</td>
<td>00:00:20</td>
<td>00:00:20</td>
<td>00:00:20</td>
<td>00:00:20</td>
</tr>
<tr>
<td>Time</td>
<td>256</td>
<td>370</td>
<td>370</td>
<td>370</td>
<td>370</td>
</tr>
</tbody>
</table>

Received 2022-11-10; accepted 2023-03-31