On the Correctness Problem for Serializability

Jürgen König[1](✉) and Heike Wehrheim[2]

[1] Paderborn University, Paderborn, Germany
jkoenig@mail.upb.de
[2] Carl von Ossietzky University of Oldenburg, Oldenburg, Germany

Abstract. Concurrent correctness conditions formalize the notion of "seeming atomicity" in concurrent access to shared object state. For different sorts of objects (databases, concurrent data structures, software transactional memory) different sorts of correctness conditions have been proposed (serializability, linearizability, opacity). Decidability of concurrent correctness conditions studies two problems: the *membership problem* asks whether a single execution is correct; the *correctness problem* asks whether all executions of a given implementation are correct.

In this paper we investigate decidability of Papadimitrious's notion of serializability for database transactions. Papadimitriou has proved the membership problem for serializability to be NP-complete. For correctness we consider a stricter version also proposed by Papadimitriou, which requires an additional real time order constraint. We show this version to be decidable given that all transactions are live.

1 Introduction

The purpose of concurrent correctness conditions is the definition of correct concurrent access to shared state. Correctness therein typically means that concurrent accesses behave as though these were happening *atomically*. Technically, this "seeming atomicity" is formalized by comparing concurrent executions (histories) to serial ones. Today, several such correctness conditions exist for varying sorts of objects, for example serializability [19] for database transactions, linearizability [16] and quiescent consistency [6] for concurrent data structures and opacity [14] for software transactional memories.

Implementations of such objects often employ intricate algorithms with fine-grained concurrency and without explicit locking. Hence, research often works towards finding model checking techniques to automatically check concurrent correctness of implementations. The quest for such techniques starts with determining the decidability and complexity of concurrent correctness conditions. Research in this area revolves around two problems: the *membership problem* and the *correctness problem*. The membership problem studies the correctness

The authors are supported by DFG grant WE2290/12-1.

© Springer Nature Switzerland AG 2021
A. Cerone and P. C. Ölveczky (Eds.): ICTAC 2021, LNCS 12819, pp. 47–64, 2021.
https://doi.org/10.1007/978-3-030-85315-0_4

of single executions, whereas the correctness problem looks at all executions generated by some implementation. In both cases, executions are compared to the behaviour of serial specifications.

In this paper, we are concerned with the correctness problem for *serializability*, the most frequently employed correctness condition for databases. Serializability was first defined by Papadimitriou [19]. Papadimitiriou has shown the membership problem to be NP-complete. For the correctness problem, Alur and McMillan [1] have studied a variant of serializability, called *conflict serializability* [10], and have shown it to be decidable and in PSPACE. Later, Bouajjani et al. [3] have shown the correctness problem for conflict serializability for an unbounded number of processes to be in EXPSPACE. Conflict serializability is based on a notion of conflict between events and (semi-)commutatibility of non-conflicting events in histories. This differs from Papadimitriou's original definition of serializability.

Here, we study decidability of the correctness problem for a definition of serializability following Papadimitriou's original idea (without a notion of conflict). More precisely, we focus on a variant of serializability, called *SSR* in [19]. *SSR* requires the reads-from relation of live transactions to be the same when comparing concurrent and serial histories as in the original definition proposed by Papadimitriou. In addition it requires the real-time order of transactions to be preserved. We prove *SSR* to be decidable under the assumption that all transactions are live. In the further we present the related work in Sect. 2, present the necessary notations and definitions in Sect. 3, our decidability result in Sect. 4, and finally give a conclusion in Sect. 5.

2 Related Work

A number of works study decidability questions for concurrent correctness conditions. A frequently studied correctness condition is conflict serializability [10]. Conflict serializability is different from (view) serializability as defined by Papadimitriou, as its equivalence definition is expressed via conflicts between events which is not possible for view serializability. Several works are concerned with the complexity of the membership and correctness problem for conflict serializability [1,3,10,19]. The correctness problem for conflict serializability is in PSPACE for a finite amount of threads [1], while for an unbounded number of threads it is EXPSPACE-complete [3]. Notably the proof for the latter result uses the fact that only a finite amount of information, independent of history length, is necessary to determine conflict serializability. Hence the basic idea to prove decidability is similarly to ours. Furthermore, multiple model checking approaches for conflict serializability have been published [5,9,11,13].

For sequential consistency, Alur and McMillan [1] have shown the correctness problem to be undecidable; a result which we used for showing undecidability of serializability. Automatic model checking techniques therefore typically work on subclasses only [15,20].

For linearizability [16], there are again results both for the membership and correctness problem [1,3,12]. Notably, the correctness problem for a bounded

$$\mathbf{R}^{1}_{t_1}(y)\mathbf{R}^{2}_{t_2}(x)\mathbf{W}^{2}_{t_2}(x)\mathbf{W}^{1}_{t_1}(x,y)\mathbf{R}^{3}_{t_1}(x)\mathbf{W}^{3}_{t_1}(z)$$
$$\mathbf{R}^{tr_w}_{t_1}()\mathbf{W}^{tr_w}_{t_1}(x,y,z)\mathbf{R}^{1}_{t_1}(y)\mathbf{R}^{2}_{t_2}(x)\mathbf{W}^{2}_{t_2}(x)\mathbf{W}^{1}_{t_1}(x,y)\mathbf{R}^{3}_{t_1}(x)\mathbf{W}^{3}_{t_1}(z)\mathbf{R}^{tr_r}_{t_1}(x,y,z)\mathbf{W}^{tr_r}_{t_1}()$$
$$\mathbf{R}^{2}_{t_2}(x)\mathbf{W}^{2}_{t_2}(x)\mathbf{R}^{1}_{t_1}(y)\mathbf{W}^{1}_{t_1}(x,y)\mathbf{R}^{3}_{t_1}(x)\mathbf{W}^{3}_{t_1}(z)$$

Fig. 1. Example histories. From top to bottom: $h_e, \overline{h_e}, h_s$.

number of threads is in EXSPACE, while the unbounded case is undecidable. Bouajjani et al. proved that for the unbounded case linearizability is still decidable for a subclass of programs that are data independent [4]. There are furthermore multiple works targetting automatic model checking of linearizability [17,21–23].

Finally, for other correctness conditions like opacity or quiescent consistency additional model checking approaches and theoretical results exist (see e.g. [2,7,8,13,18]).

3 Background

We start by defining the correctness problem for serializability. Like other concurrent correctness conditions, serializability is based on the notion of *histories*. Histories are sequences of events (reading of or writing to shared state by threads) grouped into transactions. In the sequel, we mainly follow the original definition by Papadimitriou [19].

A history is an interleaving of read and write events of a fixed number of threads. Each read and write operates on a number of variables. The events of a thread are grouped into *transactions*. A thread can execute several transactions. Thus a history is a sequence of read and write events indexed both by their threads and their transactions and parametrized by the set of accessed variables. For readability, we omit the set brackets of variable sets in examples. We let T be the finite set of threads, Var be the finite set of shared variables and Tr the set of transactions.

Definition 1 (History). *A history is a sequence of events* $ev_0 \dots ev_n$*, where for all* i*,* $0 \leq i \leq n$*, either* $ev_i = \mathbf{W}_t(V)$ *or* $ev_i = \mathbf{R}_t(V)$ *with* $t \in T, V \subseteq Var$*.*

Notation. The definition does not mention transactions since given a sequence of events indexed by threads there is – up to isomorphism – only one way to assign transaction identifiers to events – when transactions are well-formed (see below). We let $\mathcal{H}$ be the set of all histories. The set of events Ev is divided into read events Ev_{rd} and write events Ev_{wr}. The set of all transactions of a history h is $tr(h)$, and we write $tr \in h$ if a transaction tr occurs in h. Events can be indexed by their transaction tr which makes them unique, e.g. the event $\mathbf{R}^{tr}_t(V)$ is a read by thread t of all variables in V within the transaction tr. If an event occurs in a history $h \in \mathcal{H}$, we write $ev \in h$. If the event ev is ordered before another event ev' in history h, we write $ev <_h ev'$. For two histories (or more

generally sequences of events) h and h', we write $h \cdot h'$ for the concatenation of h and h', $h \preceq h'$ if h is a prefix of h', and $h \sqsubseteq h'$ if h is a subsequence of h'.

Histories have to be *well-formed* in the following sense:

1. A transaction consists of one or two events. If one, it is a read event. If two, one is a read and the other a write event.
2. Both events are executed by the same thread.
3. The write event (if it exists) is ordered after the read event, and no event of the same thread occurs in between the read and the write event.

The history h_e shown in Fig. 1 (top) is such a well-formed history.

The thread t of a transaction tr is denoted as $t(tr)$, which without loss of generality we assume to be identical for all histories, i.e., a fixed transaction tr is always executed by the same thread. We say a transaction is *unfinished* whenever it only has a read event in a history, and call it *finished* when it has two events in a history. For a transaction tr and history h we denote the first case as $unfin(tr, h)$.

For serializability we furthermore need to define real time orders as well as equivalence of histories. Two transactions tr_1, tr_2 are *real time ordered* in a history h, $tr_1 \prec_h tr_2$, when tr_1 is finished and the write event of tr_1 occurs before the read event of tr_2. The real time order of h, $h.RT \subseteq Tr \times Tr$, contains all pairs (tr_1, tr_2) such that $tr_1 \prec_h tr_2$. In h_e we for example have $1 \prec_{h_e} 3$ but $1 \not\prec_{h_e} 2$.

To define the notion of equivalence and finally serializability, we furthermore need to define (a) the reads-from relation in a history, (b) the augmentation of a history, and (c) liveness of transactions. A transaction tr_1 *reads* $v \in Var$ *from* transaction tr_2 in h whenever there exists a write event $ev = \mathbf{W}_t^{tr_2}(V)$ and a read event $ev' = \mathbf{R}_{t'}^{tr_1}(V')$ $(t, t' \in T, V, V' \subseteq Var)$ in h and $v \in V \cap V'$ such that $ev <_h ev'$ and no other event writing to v exists in between ev and ev'. The reads-from relation of h is denoted as $h.RF \subseteq Tr \times Tr \times Var$. For $tr, tr' \in tr(h)$ and $v \in Var, (tr, tr', v) \in h.RF$ means that tr' reads v from tr in h. In our example we have $(1, 3, x) \in h_e.RF$ and $(2, 3, x) \notin h_e.RF$.

To ensure that all transactions can read from some writes and all variables are read at the end, histories get augmented with additional transactions. The *augmented history* $\overline{h}$ for a history h is the history where two transactions are added, tr_w at the start and tr_r at the end of the history. The transaction tr_w writes to each variable and reads from none, and tr_r reads all variables and writes to none. For an example see the augmentation $\overline{h_e}$ of history h_e in Fig. 1 (additional transactions in grey). Then, a transaction tr in an augmented history $\overline{h}$ is called *live* whenever it either is tr_r or for a live transaction tr' and $v \in Var$, $(tr, tr', v) \in \overline{h}.RF$. A transaction is live in a non-augmented history h if it is live in its augmented version $\overline{h}$. In the example history h_e transaction 2 is not live since the only variable x it writes to is never read in $\overline{h_e}$. Note that this notion of liveness is slightly different from the notion of transaction liveness in software transactional memory (which corresponds more to being finished).

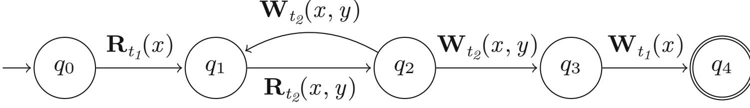

Fig. 2. Implementation Automaton Example: I_{ex}

Definition 2. *Two well-formed histories $h, h' \in \mathcal{H}$ are* equivalent *($h \equiv h'$) iff*

- *they have the same set of transactions and*
- *for any live $tr \in h$ and any $tr' \in h$, $(tr', tr, v) \in h'.RF \Leftrightarrow (tr', tr, v) \in h.RF$.*

In the example we have $h_e \equiv h_s$. As noted by Papadimitriou [19], it is actually sufficient for equivalency that both histories have the same set of *live* transactions, but w.l.o.g. this is equivalent to assuming their transactions overall are identical.

A history is *serial* whenever each read event either belongs to an unfinished transaction or is directly followed by the write of its transaction. We let $\mathcal{H}_S$ be the set of serial histories. History h_s in Fig. 1 is serial. We can now define strict serializability for histories with multiple transactions per thread. The definition mainly follows the one given by Papadimitriou[1]. Note it differs from the serializability definition employed by Alur et al. which is *conflict serializability* [10].

Definition 3 (SSR^+). *A history h is* serializable under SSR^+ *(or* strictly serializable*) iff there exists a serial history h_s such that*

1. *$h \equiv h_s$, and*
2. *$h.RT \subseteq h_s.RT$ (real time order preservation).*

Note that the real time order contains the thread order. In our example, h_s has the same real time order as h_e. Thus overall h_e is serializable under SSR^+.

Whenever for history h, a history h_s as required by the above definitions exists, we say h is *serializable to h_s under SSR^+* or call h_s an *s-witness* of h. Let h_s be a serial history and S be a set of serial histories. The set of histories serializable to h_s under SSR^+ is denoted $SSR^+(h_s)$. Additionally, $SSR^+(S)$ denotes the set of histories h such that there exists a $h_s \in S$ such that h is serializable to h_s under SSR^+.

Correctness Problem. With these definitions at hand, we can define the actual problem we are interested in. The *correctness problem* is the problem of checking whether each of the generated histories of an implementation I is serializable to some serial history generated by a specification S. We assume that both I and S – as common in the related literature [1,8] – are given as finite state automata, and let $L(A)$ be the language accepted by an automaton A. Figure 2 is

[1] The difference lays in our introduction of transaction identifiers and the accompanying requirement of thread order preservation, which is often assumed for traditional memory models.

an example of an implementation automaton. It generates (accepts) the language $L(I_{ex}) = \mathbf{R}_{t_1}(x)\left(\mathbf{R}_{t_2}(x,y)\mathbf{W}_{t_2}(x,y)\right)^{+}\mathbf{W}_{t_1}(x)$. Transaction identifiers can be freely assigned to the events of the transactions of each word in this language. We assume that both specification and implementation automaton only generate well-formed histories.

Then the correctness problem for strict serializability is defined as follows.

Problem 1 (Correctness Problem for Strict Serializability). Given an implementation I and a specification S, determine whether $L(I) \subseteq SSR^{+}(L(S))$ is true.

Assuming S to be an automaton producing every serial history (for given threads T and variables Var), the automaton I_{ex} is not correct according to the above definition. It accepts the history $\mathbf{R}^{1}_{t_1}(x)\mathbf{R}^{2}_{t_2}(x,y)\mathbf{W}^{2}_{t_2}(x,y)\mathbf{W}^{1}_{t_1}(x)$. This history is not serializable under SSR^{+}.

4 The Correctness Problem for SSR^{-} Is Decidable

We look at the decidability of the correctness problem for strict serializability. Here, we show decidability for a subclass of SSR^{+} (called SSR^{-}) where the assumption is that all transactions in a history are live or unfinished.

The decidability follows from the fact that we can construct a finite automaton whose language is empty if and only if all histories generated by the implementation automaton are strictly serializable. The states of this automaton are (approximations of) equivalence classes of histories where the equivalence captures the strict serializability of histories and their extensions.

The assumption of all transactions being live or unfinished guarantees prefix-closedness of strict serializability and thus allows us to incrementally construct the states of the equivalence class automaton.

Proposition 1. *Let $\mathcal{H}_{live}$ be the set of histories with live*[2] *transactions only, $\mathcal{H}_{un,live}$ be the set of histories where all transactions are either unfinished or live and $h \in \mathcal{H}_{live}$. If h is not strictly serializable, so are all $h' \in \mathcal{H}_{un,live}$ such that $h \preceq h'$.*

In the following we assume (1) all histories to contain live or unfinished transactions only and (2) an implementation automaton to only accept words (histories) in which all transactions are finished. We can therefore employ a notion of equivalence of histories meaning (a) same set of transactions and (b) same reads-from relation (for all transactions, not just live ones). This is important for the construction below because it allows us to directly check for the correctness of reads-from relations when observing the next read, not needing to wait for the transaction of this read to become finished. The notion of s-witness used in the sequel is based on this adapted equivalence definition.

We furthermore assume checking strict serializability against the *most general specification automaton*. The most general specification automaton generates *all*

[2] Note that all live transactions have to be finished.

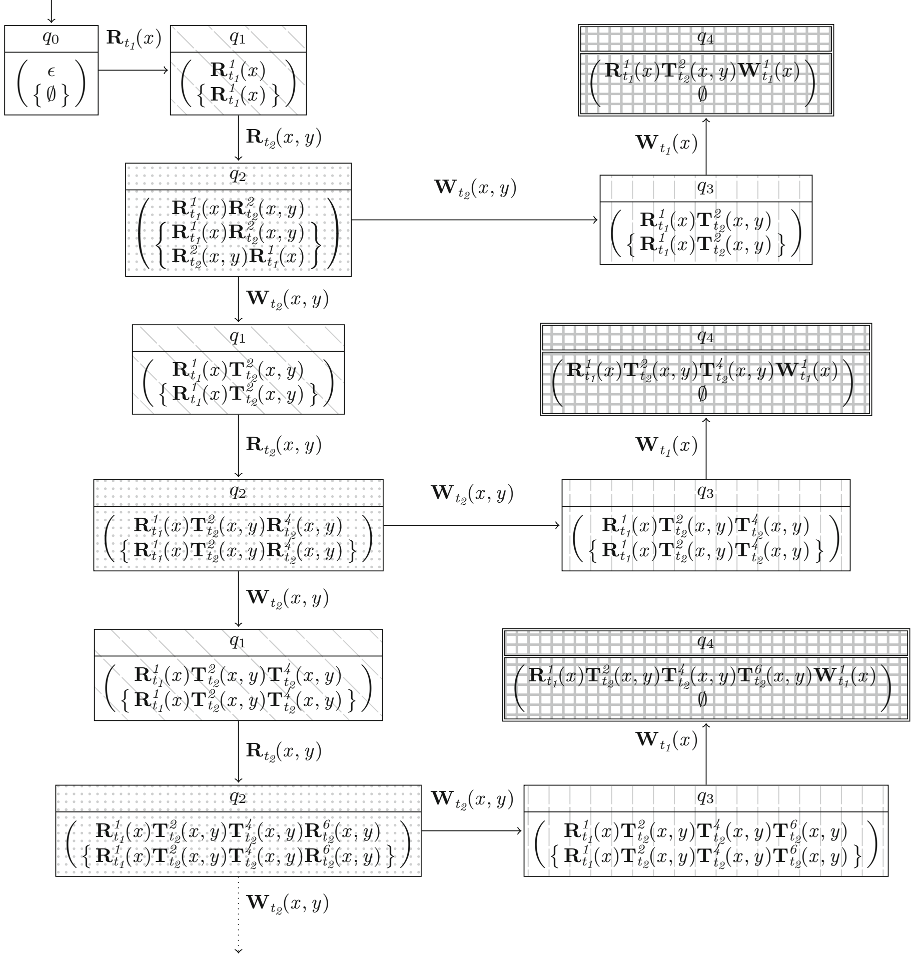

Fig. 3. Excerpt of histories of I_{ex} (of Fig. 2) and their s-witnesses

serial histories. Thus the specification automaton S does not play a role in the following. Deciding strict serializability for specific automata S would require additional tracking of states of S in the below given construction.

4.1 Compact Representation

We start by looking at a naive approach for generating all histories of an implementation automaton and explain how to compact these infinitely many histories to some finite structure. Given an implementation automaton, a naive approach would simply try to explore the entire state space of the implementation, i.e. to generate all of its histories and check them for strict serializability. An excerpt of the state space of implementation automaton I_{ex} as a graph can be seen in

Fig. 3. The upper half of each node shows the current state of the automaton and the lower half the history of the events executed so far and its set of s-witnesses. Note that an entire transaction of the form $\mathbf{R}^j_{t_i}(x)\mathbf{W}^j_{t_i}(x)$ is for brevity denoted as $\mathbf{T}^j_{t_i}(x)$.

The obvious problem with this approach is that the state space of implementations can be infinite, as there are infinitely many histories. Our approach is now to reduce the state space by *merging* nodes which behave similarly. In the graph in Fig. 3, these are marked with the same filling pattern. For example, consider the striped states (second column, second, fourth and sixth state): Whenever we execute $\mathbf{W}_{t_2}(x,y)\mathbf{W}_{t_1}(x)$ from a striped node, we end up in a node with implementation state q_4 and an empty s-witness, i.e. the current history is not strictly serializable. Whenever we execute $\mathbf{W}_{t_2}(x,y)$, we either end up in a node with implementation state q_3 or q_2 where in both cases the corresponding history is strictly serializable. So summarizing we consider two nodes as behaving similarly whenever

- they contain the same implementation automaton state, and
- when appending identical events, both either keep or loose their strict serializability.

Merging these two nodes into one does not change the accepted language of the automaton. We show decidability by proving that such a graph with merged nodes has (a) a finite number of nodes (and thus is representable as a finite automaton) and (b) this automaton is effectively constructable.

We start by formalizing the above similarity on histories.

Definition 4 (SSR-extension equivalence). *Two histories $h, h' \in \mathcal{H}$ are SSR-extension equivalent ($h \equiv_{ext} h'$) iff $\forall n \in \mathbb{N}, \forall ev_0 \dots ev_n \in Ev^n$ either*

- *$h \cdot ev_0 \dots ev_n$ and $h' \cdot ev_0 \dots ev_n$ are both strictly serializable,*
- *or $h \cdot ev_0 \dots ev_n$ and $h' \cdot ev_0 \dots ev_n$ are both not strictly serializable.*

The question is how to determine whether two histories are SSR-extension equivalent. The general idea is to reduce a history to the essential information needed to determine whether appending events keeps the history strictly serializable or not. This information is called *SSR-data*. Whenever two histories have the same SSR-data, they are SSR-extension equivalent. Below we will show that there are only finitely many different (valid) SSR-data which is key to our decidability result.

Witness Extensions. Before we formalize SSR-data, we take a look at some properties of histories, their s-witnesses and extensions with events. Figure 4 shows the first such property in a diagram. The upper level is a history h and its extension h' by one read event. The read event is (and can only be) appended at the end. The lower level depicts one s-witness (h_s) for h and one for h' (h'_s). Here, we see that the new read event is inserted in the middle of h_s. What is important, however, is that the new s-witness h'_s needs to be a *supersequence*

of h_s, i.e. we cannot reorder events already occurring in h_s. This is due to the requirements of strict serializability (equality of reads-from relation and subset on real-time ordering). The same applies to extensions with write events. Hence, we only have a limited amount of candidate s-witnesses when extending a history.

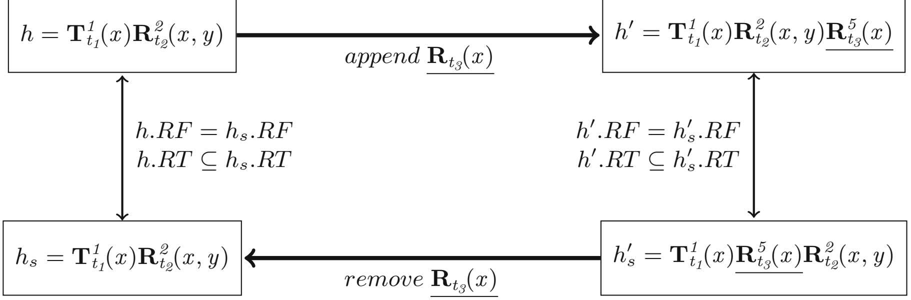

Fig. 4. Supersequence property for extensions of s-witnesses

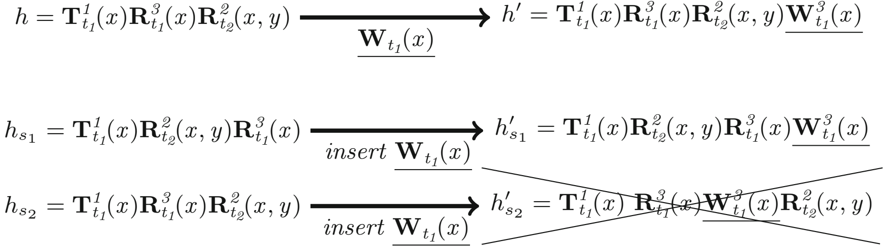

Fig. 5. A history and its s-witnesses extended with a write event

Next, we need to be able to compute s-witness extensions (or at least, a compact form of them). To this end, we need to determine which of the supersequence candidates can be kept and which are to be eliminated because they are no valid s-witnesses for the extended history. For this, consider Fig. 5. The history h (top) is strictly serializable and both the histories h_{s_1} and h_{s_2} are s-witnesses. When history h is extended by event $\mathbf{W}^{3}_{t_1}(x)$, this event gets appended at the end of the history. Similarly, we need to insert $\mathbf{W}^{3}_{t_1}(x)$ into the s-witnesses to find a witness for h'.

As this is a write event, all serial candidates must have the write by transaction 3 directly follow the read event of 3. In Fig. 5 the write is thus inserted directly after the last event of its thread in both cases. The resulting histories are obviously serial. Second, we need to check whether h'_{s_1} and h'_{s_2} preserve the real time order of h'. This is the case. Third, we need to check if the reads-from orders of h' and h'_{s_1}, h'_{s_2}, respectively, are identical. For h'_{s_1} this is the case as well. For h'_{s_2} they are different: In h_{s_2} transaction 1 writes to x which transaction 2 reads with transaction 3 occuring in between. We say that x belongs to

the *write-before-read-after* (short, *wbra*) *variables* of 3. Thus in h'_{s_2} the write of transaction 3 is read (by transaction 2) which it is not in h'. Hence h'_{s_2} is not an s-witness of h' and this candidate needs to be eliminated.

The elimination can be determined by looking at the write-before-read-after variables formalized with the help of the last-writer function.

Definition 5 (Last Writer). *Given a history* $h = ev_0 \dots e_n$*, the* last writer function *for an event* ev*,* $lw_{ev,h} : Var \rightarrow Tr \cup \{tr_{ini}\}$[3]*, determines the last writer to a variable* v*, i.e.* $lw_{ev,h}(v) = tr$ *iff*

- $tr \in h$ *and* tr *contains a write event* w *writing to* v*,*
- $w <_h ev$ *(the write occurs before* ev*),*
- *and there is no write* w' *to* v *such* $w <_h w' <_h ev$*.*

Note that it is not possible to use the reads-from relation here, as the last writer function returns the last writer of a variable at arbitrary specified event of the history, which does not have to be a read, reading that variable. When the event ev *is the last event in a history, we elide the index to* lw*.*

The *write-before-read-after variables* of a transaction are all variables that get written to before and read from after that transaction.

Definition 6 (Write-Before-Read-After Variables). *Given a serial history* h_s *with unfinished transaction* tr *and read event* ev*, the* write-before-read-after function $wbra_h : Tr \rightarrow 2^{Var}$ *of* h *determines the variables written before and read after a transaction, i.e.* $v \in wbra_h(tr)$ *holds iff*

- *there exists a transaction* tr_1 *s.t.* $lw_{ev,h}(v) = tr_1$ *and*
- *a transaction* tr_2 *with a read event* ev' *s.t.* $ev <_h ev'$ *and* $lw_{ev',h}(v) = tr_1$*.*

If tr *is finished, then* $wbra_h(tr) = \emptyset$*.*

Summarizing, for extensions with write events we get the following property: an s-witness h_s can be extended with a write event $\mathbf{W}_t^{tr}(V)$ if

$$wbra_{h_s}(tr) \cap V = \emptyset \ . \tag{1}$$

Next we look at extensions with read events. For a read the preservation of the serial nature of an s-witness is trivial, as a new read does not violate it no matter where it is added. Still all feasible candidates must have the additional read located after the last write of the old s-witness. Otherwise the real time order of the extended history is trivially not preserved. In Fig. 6 there are two s-witnesses for h with different orders for the writing transactions. Note that in every history except h we removed all empty reads for brevity. For h_{s_1} two successor candidates exist, h'_{s_1} where the read is added after $\mathbf{R}_{t_1}^{3}(x)$ and h'_{s_2} where it is added before. Similary for h_{s_2} two candidates exist. For both s-witnesses both successor candidates preserve the real time order of h'.

[3] We assume tr_{ini} to be the transaction initializing all variables.

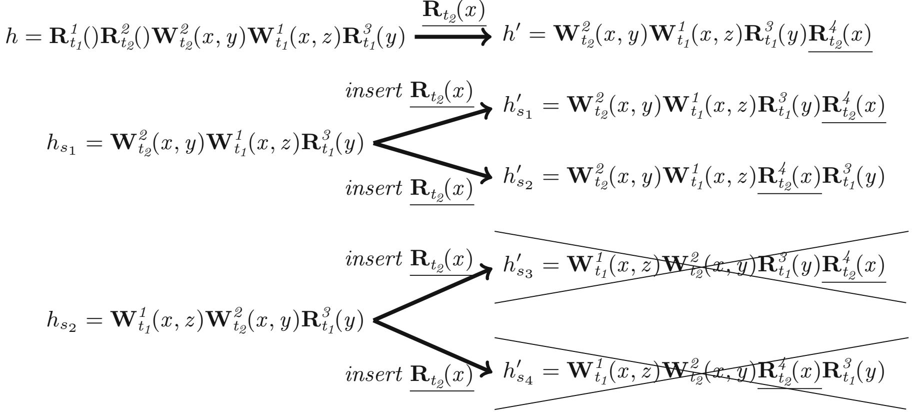

Fig. 6. A history and its s-witnesses extended with a read event

For the reads-from relation both successor candidates for h_{s_1} are valid as the new read event reads from transaction 1 in both cases. The candidates for h_{s_2} differ in reads-from order, as the read event reads from transaction 2. This is the case since in h_{s_1} the last writer on x is transaction 1 which is identical to that of h, but for h_{s_2} transaction 2 is the last writer which is different from h. The summary in such and similar cases is thus: an s-witness h_s can be extended with a read event $\mathbf{R}_t^{tr}(V)$ if for all variables $v \in V$, the last writer of v in h and h_s is the same:

$$\forall v \in V : lw_{h_s}(v) = lw_h(v). \tag{2}$$

These considerations lead us to keeping both the last writer and the wbra variables in the SSR-data.

SSR-Data. We can now take a look at the SSR-data and its extension for events for an example (Fig. 7). Note that in the full state space of an implementation automaton we would need to store a complete history and its set of s-witnesses. Here we now apply two compression functions to the history and to each s-witness, and only store their compressed versions together with the wbra variables. The compression works as follows: For each s-witness we remove every finished transaction that is not a last writer; for the history we remove each transaction that is finished and not a last writer in any (compressed) s-witness of the history. The first compression function is denoted as *sub* ($sub : \mathcal{H} \to \mathcal{H}$), the latter as *suball* ($suball : \mathcal{H} \times 2^{\mathcal{H}} \to \mathcal{H}$). In $suball(h, H)$, the set H is some set of (possibly already compressed) s-witnesses. Both compression functions generate strings which are subsequences of their (first) argument.

In the given example history $\mathbf{T}_{t_2}^{2}(x)\mathbf{T}_{t_1}^{1}(x)$ (Fig. 7, top node) the first transaction is not the last writer of any variable, it is also finished, so it is removed when extracting SSR-data. The *wbra* variables are shown as "–" as there is no

unfinished transaction. In the one s-witness, transaction 1 is removed as it is finished. Then we see a number of extensions with events (transitions from left to right) followed by compression steps (diagonal arrows from right to left). These show how the SSR-data is first extended and then again compressed. When a new event is appended to the history each compressed s-witness is expanded like a normal s-witness, as discussed above. As all last writers are known (can be seen from compacted s-witnesses and history) we can compare them. We can also check whether a write is in conflict with the *wbra* variables of its transaction (condition (1)). For each new s-witness the wbra set is generated from the previous s-witness. After each extension the resulting tuple is compressed again.

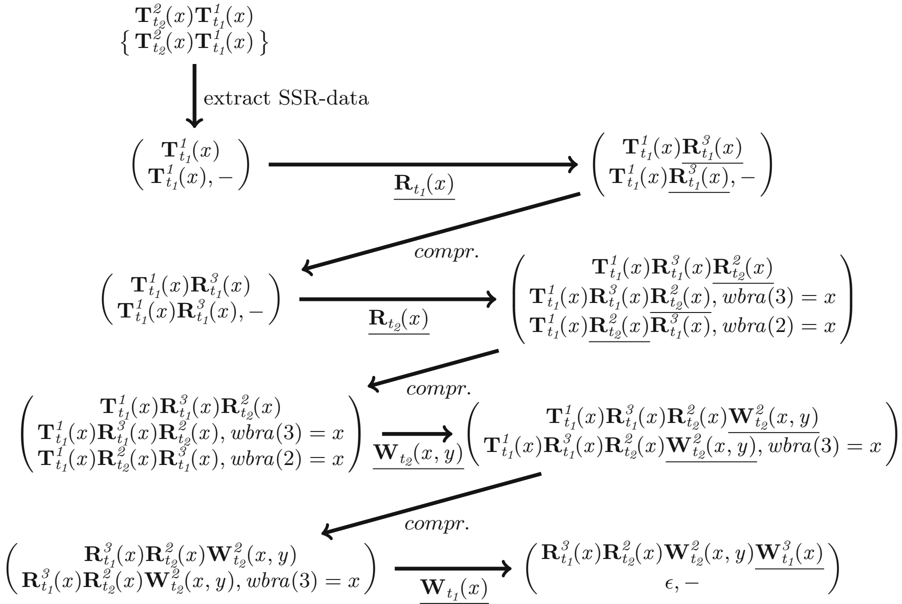

Fig. 7. Examples for the computation of successors of SSR-data; *wbra* variables only shown if non-empty.

In general SSR-data are elements of the form $\mathcal{H} \times 2^{\mathcal{H}_S \times W}$, where W is the set of all functions $wbra : Tr \rightarrow Var$. We store a (compressed) history together with a set of pairs containing (compressed) s-witnesses and their wbra functions.

Definition 7 (Validity of SSR-data). *Let $h \in \mathcal{H}$ be a history and H_s its set of s-witnesses. A pair $(h_c, HW) \in \mathcal{H} \times 2^{\mathcal{H}_S \times W}$ is* valid SSR-data for *h iff*

- *$h_c = suball(h, H_s)$ (compressed history), and*
- *$HW = \{(sub(h_s), wbra_{h_s}) \mid h_s \in H_s\}$ (pairs of compressed s-witnesses and their wbra functions).*

Proposition 2. *Let $h \in \mathcal{H}$ not be strictly serializable. Then its valid SSR-data is $(suball(h, \emptyset), \emptyset)$ where $suball(h, \emptyset) \neq \epsilon$.*

Key to decidability is the fact that we only have a finite amount of different valid SSR-data.

Lemma 1. *The number of SSR-data valid for some history is bound in size by $O((|Var| \cdot 2^{2|Var|} + |T| \cdot 2^{|Var|})!)$.*

We next formally define the successor computation (as in Fig. 7). The extension and compression step are unified into one function, which is composed out of a function for appending a write and one for appending a read event ev.

$$ext((h_c, HW), ev) = \begin{cases} ext_r((h_c, HW), ev) & \text{if } ev \in Ev_{rd}, \\ ext_w((h_c, HW), ev) & \text{if } ev \in Ev_{wr}. \end{cases}$$

We next define ext_w and ext_r starting with write extensions. For each compressed s-witness of the input SSR-data, we need to check whether the writing thread's *wbra* variables contain any of the variables written to by the write event (Condition (1)). If not, then both (compressed) s-witness and (compressed) history need to be extended with the write event and the wbra variables of all transactions updated. Let $ev(tr)$ denote the last element of transaction tr in the context of a history.

Definition 8 (Extension with write). *Let (h_c, HW) be some SSR-data and $ev = \mathbf{W}_t^{tr}(V)$ a write event.*
Then $ext_w((h_c, HW), ev) = (h'_c, HW')$ where $(h'_s, wb_{h'_s}) \in HW'$ iff there exists some pair $(h_s, wb_{h_s}) \in HW$ such that $h_s = ev_0 \dots ev(tr) \dots ev_n$ and

- *$wb_{h_s}(tr) \cap V = \emptyset$ (no writing of wbra variables),*
- *$h'_s = sub(ev_0 \dots ev(tr)ev \dots ev_n)$ (compression of extended s-witness),*
- *$wb_{h'_s}(tr) = \emptyset$ (wbra variables of finished transaction emptied),*
- *$\forall tr' \neq tr \in Tr : wb_{h'_s}(tr') = wb_{h_s}(tr')$ (wbra variables of other transactions kept)*

and $h'_c = suball(h_c \cdot ev, HW')$ (history compressed w.r.t. new s-witnesses).

This write extension preserves validity of SSR-data.

Lemma 2. *Let h be a history, (h_c, HW) its valid SSR-data and ev a write event. Then the SSR-data $ext_w((h_c, HW), ev)$ is valid for $h \cdot ev$.*

Next we define the extension with read events. For each s-witness we check if its last writers for the variables read are identical with that of the compressed history (condition (2)); if yes we generate all candidates where the new read is placed after the last write. We then compress these and update the *wbra* variables. Finally the compressed history is expanded and again compressed using the information from the new s-witness set. Let $lwr(h)$ denote the last write event of history h.

Definition 9 (Extension with reads). *Let* (h_c, HW) *be some SSR-data and* $ev = \mathbf{R}_t^{tr}(V)$ *a read event.*
Then $ext_r((h_c, HW), ev) = (h'_c, HW')$ *where* $(h'_s, wb_{h'_s}) \in HW'$ *iff there exists some pair* $(h_s, wb_{h_s}) \in HW$ *such that* $h_s = ev_0 \dots lwr(h_s) \dots ev_n$ *and*

- $\forall v \in V : lw_{h_s}(v) = lw_{h_c}(v)$ *(last writers of history and s-witnesses agree),*
- $h'_s = sub(ev_0 \dots lwr(h_s) \dots ev \dots ev_n)$ *(compression of extended s-witness, read inserted somewhere after last write),*
- $\forall tr' \in Tr : wb_{h'_s}(tr') = wb_{h_s}(tr') \cup wbra_{h'_s}(tr')$ *(wbra variables of all transactions updated)*

and $h'_c = suball(h_c \cdot ev, HW')$*.*

This read extension preserves validity of SSR-data.

Lemma 3. *Let* h *be a history,* (h_c, HW) *its valid SSR-data and* ev *a read event. Then the SSR-data* $ext_r((h_c, HW), ev)$ *is valid for* $h \cdot ev$*.*

For a sequence *seq* and some SSR-data (h_c, HW), we write $ext((h_c, HW), seq)$ for the consecutive extension of the SSR-data with the events of *seq*. Now given an event sequence h, we can simply apply *ext* consecutively for each event, and if none of the thus computed SSR-data contains $\emptyset$ as the second element of the pair, the history h is strictly serializable.

SSR-Data and SSR-Extension Equivalence. As a last step in the definition of SSR-data, we show the desired property about SSR-extension equivalence: if two histories h and h' have the "same" valid SSR-data, then they are SSR-extension equivalent. Here we employ similarity up to transaction renamings, i.e. transaction identifiers can be arbitrarily renamed via a bijective function $r : Tr \to Tr$ when r preserves threads (for all tr, $t(tr) = t(r(tr))$). We write $(h_c, HW) \equiv_{data} (h'_c, HW')$ if the two SSR-data are the same up to renaming of transactions.

Theorem 1. *Let* $h, h' \in \mathcal{H}$ *be two histories and* (h_c, HW) *and* (h'_c, HW') *their valid SSR-data. If* $(h_c, HW) \equiv_{data} (h'_c, HW')$*, then* $h \equiv_{ext} h'$*.*

The reverse implication does not hold: SSR-data is only approximating SSR-extension equivalence. For the correctness of the automaton construction given next this direction of the implication suffices.

4.2 Construction of Finite Automaton

In our decision procedure, we generate a finite automaton where the states are pairs of implementation automaton states and SSR-data of histories. The infinite state space obtained via the naive exploration strategy is thus collapsed into a finite automaton. This automaton can be constructed by starting with the SSR-data of an empty history and then generating new SSR-data according to the events in the implementation automaton using the above given extension

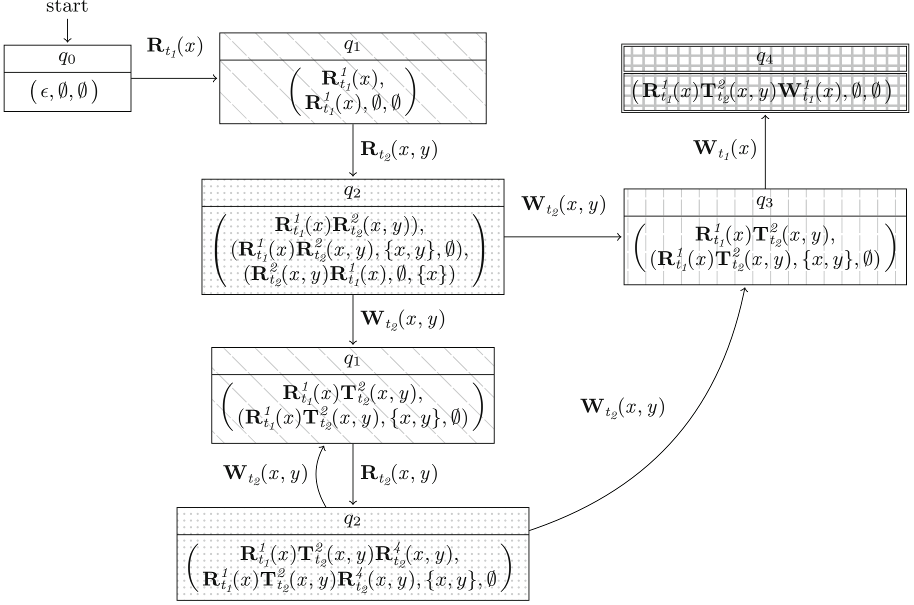

Fig. 8. SSR-automaton of I_{ex} (of Fig. 2)

function. As we only have a finite number of different SSR-data (as of Lemma 1) such a construction terminates.

To formalize this construction, we let $SSR_{T,Var}$ be the set of all SSR-data with thread identifiers from T and variables from Var. We furthermore let $SSR\emptyset_{T,Var}$ be the set of all SSR-data of the format $(h_c, \emptyset)$, where $h_c \neq \epsilon$.

Definition 10. *Let $I = (Q, \delta, q_0, F)$ be an implementation automaton. The* SSR-automaton of *I ($E(I)$) is the automaton $(Q_E, \delta_E, q_{0,E}, F_E)$ such that*

- $Q_E = Q \times SSR_{T,Var}$,
- $q_{0,E} = (q_0, (\epsilon, \emptyset))$,
- $F_E = F \times SSR\emptyset_{T,Var}$

and $((q, ssr), ev, (q', ssr')) \in \delta_E$ iff $(q, ev, q') \in \delta$ and $ext(ssr, ev) = ssr'$.

The thus constructed automaton is a finite automaton since by Lemma 1 we only have finitely many different valid SSR-data. Furthermore, we can derive strict serializability of the implementation automaton from the language of the SSR-automaton.

Theorem 2. *Let I be an implementation automaton. Then I is strictly serializable iff $L(E(I)) = \emptyset$.*

This finally gives us the decidability of SSR^-.

Corollary 1. *The correctness problem for SSR^- is decidable.*

Figure 8 shows the result of the construction for our running example. The diagram only depicts the reachable states. Note that the standardized naming of transactions can lead to a "renaming" of transactions and does so for transaction 3 in one case. We see that the language of the SSR-automaton is non-empty (the state with the grid pattern is accepting), and hence not all histories of the implementation automaton are strictly serializable. We also see that equivalence of SSR-data only implies SSR-extension equivalence: there are still two striped and two dotted states which are SSR-extension equivalent but have different SSR-data, and thus could not be compacted to a single state.

5 Conclusion

In this paper we have studied the decidability of the correctness problem of serializability. We have proven a strenghtening of serializability with an additional requirement of real-time order preservation to be decidable. As future work we plan to investigate whether our assumption of liveness of transactions can be removed while keeping the decidability result.

References

1. Alur, R., McMillan, K.L., Peled, D.A.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. **160**(1–2), 167–188 (2000). https://doi.org/10.1006/inco.1999.2847
2. Armstrong, A., Dongol, B., Doherty, S.: Reducing opacity to linearizability: a sound and complete method. CoRR abs/1610.01004 (2016). http://arxiv.org/abs/1610.01004
3. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 290–309. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_17
4. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On reducing linearizability to state reachability. Inf. Comput. **261**(Part), 383–400 (2018). https://doi.org/10.1016/j.ic.2018.02.014
5. Cohen, A., O'Leary, J.W., Pnueli, A., Tuttle, M.R., Zuck, L.D.: Verifying correctness of transactional memories. In: Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, 11–14 November 2007, Proceedings, pp. 37–44. IEEE Computer Society (2007). https://doi.org/10.1109/FAMCAD.2007.40
6. Derrick, J., Dongol, B., Schellhorn, G., Tofan, B., Travkin, O., Wehrheim, H.: Quiescent consistency: defining and verifying relaxed linearizability. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 200–214. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_15
7. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Aspects Comput. **25**(5), 769–799 (2013). https://doi.org/10.1007/s00165-012-0225-8

8. Dongol, B., Hierons, R.M.: Decidability and complexity for quiescent consistency. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, 5–8 July 2016, pp. 116–125. ACM (2016). https://doi.org/10.1145/2933575.2933576
9. Emmi, M., Majumdar, R., Manevich, R.: Parameterized verification of transactional memories. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, 5–10 June 2010, pp. 134–145. ACM (2010). https://doi.org/10.1145/1806596.1806613
10. Eswaran, K.P., Gray, J., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM **19**(11), 624–633 (1976). https://doi.org/10.1145/360363.360369
11. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_8
12. Gibbons, P.B., Korach, E.: The complexity of sequential consistency. In: Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, SPDP 1992, Arlington, Texas, USA, 1–4 December 1992, pp. 317–325. IEEE Computer Society (1992). https://doi.org/10.1109/SPDP.1992.242728
13. Guerraoui, R., Henzinger, T.A., Singh, V.: Model checking transactional memories. Distrib. Comput. **22**(3), 129–145 (2010). https://doi.org/10.1007/s00446-009-0092-6
14. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Chatterjee, S., Scott, M.L. (eds.) Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, 20–23 February 2008, pp. 175–184. ACM (2008). https://doi.org/10.1145/1345206.1345233
15. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Verifying sequential consistency on shared-memory multiprocessor systems. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 301–315. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_27
16. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
17. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_21
18. O'Leary, J.W., Saha, B., Tuttle, M.R.: Model checking transactional memory with spin. In: 29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), Montreal, Québec, Canada, 22–26 June 2009, pp. 335–342. IEEE Computer Society (2009). https://doi.org/10.1109/ICDCS.2009.72
19. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979). https://doi.org/10.1145/322154.322158
20. Qadeer, S.: Verifying sequential consistency on shared-memory multiprocessors by model checking. IEEE Trans. Parallel Distrib. Syst. **14**(8), 730–741 (2003). https://doi.org/10.1109/TPDS.2003.1225053
21. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_40

22. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 261–278. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02652-2_21
23. Zhang, S.J.: Scalable automatic linearizability checking. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, 21–28 May 2011, pp. 1185–1187. ACM (2011). https://doi.org/10.1145/1985793.1986037