Acta Informatica (2018) 55:649–668
https://doi.org/10.1007/s00236-017-0305-6

CrossMark

ORIGINAL ARTICLE

TSO-to-TSO linearizability is undecidable

Chao Wang[1,2] · **Yi Lv**[1,2] · **Peng Wu**[1,2]

Received: 30 May 2016 / Accepted: 4 October 2017 / Published online: 23 October 2017
© Springer-Verlag GmbH Germany 2017

Abstract *TSO-to-TSO linearizability* is a variant of linearizability for concurrent libraries on the total store order (TSO) memory model. It is proved in this paper that TSO-to-TSO linearizability for a bounded number of processes is undecidable. We first show that the trace inclusion problem of a classic-lossy single-channel system, which is known undecidable, can be reduced to the history inclusion problem of specific libraries on the TSO memory model. Based on the equivalence between history inclusion and extended history inclusion for these libraries, we then prove that the extended history inclusion problem of libraries is undecidable on the TSO memory model. By means of extended history inclusion as an equivalent characterization of TSO-to-TSO linearizability, we finally prove that TSO-to-TSO linearizability is undecidable for a bounded number of processes. Additionally, we prove that all variants of history inclusion problems are undecidable on TSO for a bounded number of processes.

1 Introduction

Libraries of high performance concurrent data structures have been widely used in concurrent programs to take advantage of multi-core architectures, such as *java.util.concurrent* for Java and *std::thread* for C++11. It is important but notoriously difficult to ensure that concurrent libraries are designed and implemented correctly. *Linearizability* [13] is accepted as a de

This work is partially supported by the National Natural Science Foundation of China under Grants Nos. 60721061, 60833001, 61672504, 61572478, 61672503, 61100069, 61161130530, the National Key Basic Research Program of China under Grant No. 2014CB340701, and the National Key Research and Development Program of China under Grant No. 2017YFB0801900.

✉ Chao Wang
wangch@ios.ac.cn

✉ Yi Lv
lvyi@ios.ac.cn

1 State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

2 University of Chinese Academy of Sciences, Beijing, China

facto correctness condition for a concurrent library with respect to its sequential specification on the sequential consistency (SC) memory model [14]. Intuitively, linearizability provides the vision that every individual operation appears to take place instantaneously at some point between its invocation and return. It is well known that on the SC memory model linearizability of a concurrent library is decidable for a bounded number of processes [1], but undecidable for an unbounded number of processes [6].

However, modern multiprocessors (e.g., 86 $\times$ [17], POWER [18]) and programming languages (e.g., C/C++ [5], Java [16]) do not comply with the SC memory model. As a matter of fact, they provide *relaxed memory models*, which allow subtle behaviors due to hardware or compiler optimization. For instance, in a multiprocessor system implementing the total store order (TSO) memory model [17], each processor is equipped with a FIFO store buffer. Any write action performed by a processor is put into its local store buffer first and can then be flushed into the main memory at any time.

The notion of linearizability has been extended for relaxed memory models, e.g., *TSO-to-TSO linearizability* [8] and *TSO-to-SC linearizability* [12] for the TSO memory model and two variants of linearizability [3] for the C++ memory model. These notions generalize the original one by relating concurrent libraries with their abstract implementations, in the way as shown in [11] for the SC memory model. It is worth mentioning that these notions of linearizability satisfy the abstraction theorem [3,8,12]: if a library is linearizable with respect to its abstract implementation, every observable behavior of any client program using the former can be observed when the program uses the latter instead. Concurrent software developer can benefit from this correspondence in that the library can be safely replaced with its abstract implementation for the sake of optimization or the ease of verification of the client program.

The decision problems for linearizability on relaxed memory models become more complicated. Because of the hierarchy of memory models, it is rather trivial to see that linearizability on relaxed memory models is undecidable for an unbounded number of processes, based on the known undecidability result on the SC memory model [6]. But the decision problem of linearizability on relaxed memory models remains open for a bounded number of processes.

In this paper we mainly study the decision problem for the TSO-to-TSO linearizability of concurrent libraries within a bounded number of processes. TSO-to-TSO linearizability is the first definition of linearizability on relaxed memory models. It relates a library running on the TSO memory model to its abstract implementation running also on the TSO memory model. Histories of method invocations/responses are typically concerned by the standard notion of linearizability. For TSO-to-TSO linearizability, such histories have to be extended to reflect the interactions between concurrent libraries and processor-local store buffers.

The main result of this paper is that TSO-to-TSO linearizability is undecidable for a bounded number of processes. We first show that the extended history inclusion is an equivalent characterization of TSO-to-TSO linearizability. Then, we prove our undecidability result by reducing the trace inclusion problem between any two configurations of a classic-lossy single-channel system to the extended history inclusion problem between two specific libraries. Recall that the trace inclusion problem between configurations of a classic-lossy single-channel system is undecidable [19]. The reduction is achieved by using as a bridge the history inclusion between these two specific libraries.

Technically, we present a library template that can be instantiated as a specific library for a configuration of a classic-lossy single-channel system. The library is designed with three methods M_i for $1 \leq i \leq 3$. We use two processes P_1 and P_2, calling methods M_1 and

M_2, respectively, to simulate the traces of the classic-lossy single-channel system starting from the given configuration. This is based on the observation that on the TSO memory model, a process may miss updates by other processes because multiple flush actions may occur between consecutive read actions of the process [2]. But a channel system accesses the content of a channel always in a FIFO manner; while on the contrary, a process on the TSO memory model always reads the latest updates in its local store buffer (whenever possible). Herein, processes P_1 and P_2 alternatively update their own store buffers, but read only from each other's store buffer. In this way, the labeled transitions of the classic-lossy single-channel system can be reproduced through the interactions between processes P_1 and P_2. Furthermore, we use the third process P_3, calling method M_3 repeatedly, to return each fired transition label repeatedly, so that the traces of the classic-lossy single-channel system starting from a given configuration can be mimicked by the histories of the library exactly. Specially, methods M_1 and M_2 never return, while method M_3 just uses an atomic write action to return labels in order not to touch process P_3's store buffer. Consequently, we can easily establish the equivalence between the history inclusion and the extended history inclusion between the specific libraries.

By constructing two specific libraries based on the above library template, we show that the trace inclusion problem between any two configurations of a classic-lossy single-channel system can be reduced to the history inclusion problem between the corresponding two concurrent libraries, while the history inclusion relation and the extended history inclusion relation are equivalent between these two libraries. Then, the undecidability result of TSO-to-TSO linearizability for a bounded number of processes follows from its equivalent characterization and the undecidability result of classic-lossy single-channel system. To our best knowledge, this is the first result on the decidability of linearizability of concurrent libraries on relaxed memory models.

Apart from histories and extended histories, there are other forms of sequences that are used to represented behaviors of libraries. For example, in [9,10] the behavior of concurrent libraries on TSO are essentially recorded by sequences of call and flush return actions. Based on this variant of history, they propose TSO-linearizability, a variant of linearizability without abstraction theorem, as correctness condition. To deal with various possible forms of histories, we also consider variants of histories. As by-product of our work, we prove that all variants of history inclusion problems, including history inclusion problem and extended history inclusion problem, are undecidable on TSO for a bounded number of processes. Some variants of history inclusion problems can be similarly proved as history inclusion problem and extended history inclusion problem. To deal with other variants of history inclusion problems, we slightly modify the specific libraries. Then the traces of the classic-lossy single-channel system can be mimicked by sequences of call actions.

Related work Efforts have been devoted on the decidability and model checking of linearizability on the SC memory model [1,6,7,15,20]. The principle of our equivalent characterization for TSO-to-TSO linearizability is similar to that of the characterization given by Bouajjani *et al.* in [7], where history inclusion is proved to be an equivalent characterization of linearizability. Alur *et al.* proved that for a bounded number of processes, checking whether a regular set of histories is linearizable with respect to its regular sequential specification can be reduced to a history inclusion problem, and hence is decidable [1]. Bouajjani *et al.* proved that the problem of whether a library is linearizable with respect to its regular sequential specification for a unbounded number of processes is undecidable, by a reduction from the reachability problem of a counter machine (which is known to be undecidable) [6].

On the other hand, the decidability of linearizability on relaxed memory models is still open for a bounded number of processes. The closest work to ours is [2] by Atig *et al.*, where a lossy channel system is simulated by a concurrent program on the TSO memory model. Our approach of using methods M_1 and M_2 to simulate a classic-lossy single-channel system is inspired by their work. However, in [2], it was the decidable reachability problem of the channel system that was reduced to the reachability problem of the concurrent program on the TSO memory model. Hence, only the start and end configurations of the channel system are needed in their reduction. In this paper, we reduce the trace inclusion problem between any two configurations of a classic-lossy single-channel system, which is undecidable, to the TSO-to-TSO linearizability problem. Our reduction needs to show exactly each step of transitions in the channel system.

Paper outline We give the definitions of libraries, concurrent systems and its operational semantics in Sect. 2. We introduce the definition of TSO-to-TSO linearizability and variants of histories, and prove that extended history inclusion is an equivalent characterization of TSO-to-TSO linearizability in Sect. 3. In Sect. 4, we present how to generate specific libraries to mimic behaviors of classic-lossy single-channel systems. We prove in Sect. 5 that TSO-to-TSO linearizability and all variants of history inclusion problems are undecidable on TSO for a bounded number of processes. We conclude in Sect. 6.

Differences from the conference paper This article is an extended version of our ATVA'15 conference paper [22], containing all the proofs of the lemmas and theorems mentioned in the paper. Since the conference, we have also extended our result from undecidability of history inclusion problem and extended history inclusion problem into all variants of history inclusion problems.

2 TSO concurrent systems

In this section, we first present the notations of libraries, the most general clients and TSO concurrent systems. Then, we introduce their operational semantics on the TSO memory model.

2.1 Notations

In general, a finite sequence on an alphabet Σ is denoted $l = \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_k$, where $\cdot$ is the concatenation symbol and $\alpha_i \in \Sigma$ for each $1 \leq i \leq k$. Let $|l|$ denote the length of l, i.e., $|l| = k$, and $l(i)$ denote the ith element of l for $1 \leq i \leq k$, i.e., $l(i) = \alpha_i$. For an alphabet Σ', let $l \uparrow_{\Sigma'}$ denote the projection of l to Σ'. Given a function f, let $f[x : y]$ be the function that shares the same value as f everywhere, except for x, where it has the value y. We use _ for an item, of which the value is irrelevant.

A *labelled transition system* (LTS) is a tuple $\mathcal{A} = (Q, \Sigma, \rightarrow, q_0)$, where Q is a set of states, Σ is a set of transition labels, $\rightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation and q_0 is the initial state. A state of the LTS $\mathcal{A}$ may be referred to as a *configuration* in the rest of the paper.

A path of $\mathcal{A}$ is a finite transition sequence $q_1 \xrightarrow{\beta_1} q_2 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_k} q_{k+1}$ for $k \geq 0$. A trace of $\mathcal{A}$ is a finite sequence $t = \beta_1 \cdot \beta_2 \cdot \ldots \cdot \beta_k$, where $k \geq 0$ if there exists a path $q_1 \xrightarrow{\beta_1} q_2 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_k} q_{k+1}$ of $\mathcal{A}$. Let $path(\mathcal{A}, q)$ and $trace(\mathcal{A}, q)$ denote all the paths and traces of $\mathcal{A}$ that start from q, respectively. We write $path(\mathcal{A})$ and $trace(\mathcal{A})$ for short if $q = q_0$.

2.2 Libraries and the most general clients

A library implementing a concurrent data structure provides a set of methods for external users to access the data structure. It may contain private memory locations for its own use. A client program is a program that interacts with libraries. For simplicity, we assume that each method has just one parameter and one return value if it returns. Furthermore, all the parameters and the return values are passed via a special register r_f.

For a library, let $\mathcal{X}$ be a finite set of its memory locations, $\mathcal{M}$ be a finite set of its method names, $\mathcal{D}$ be its finite data domain, $\mathcal{R}$ be a finite set of its register names and $\mathcal{RE}$ be a finite set of its register expressions over $\mathcal{R}$. Then, a set *PCom* of primitive commands considered in this paper includes:

- Register assign commands in the form of $r_1 = re$;
- Register reset commands in the form of *havoc*;
- Read commands in the form of $read(x, r_1)$;
- Write commands in the form of $write(r_1, x)$;
- Lock commands in the form of *lock*;
- Unlock commands in the form of *unlock*;
- Assume commands in the form of $assume(r_1)$;
- Call commands in the form of $call(m)$;

where $r_1 \in \mathcal{R}$, $re \in \mathcal{RE}$, $x \in \mathcal{X}$. Herein, the commands in this paper are borrowed from [8], where lock and unlock commands are called xlock and xunlock in [8]. A *havoc* command [8] assigns arbitrary values to all registers in $\mathcal{R}$.

A control-flow graph is a tuple $CFG = (N, L, T, q_i, q_f)$, where N is a finite set of program positions, L is a set of primitive commands, $T \subseteq N \times L \times N$ is a control-flow transition relation, q_i is the initial position and q_f is the final position.

A library $\mathcal{L}$ can then be defined as a tuple $\mathcal{L} = (Q_{\mathcal{L}}, \rightarrow_{\mathcal{L}}, InitV_{\mathcal{L}})$, such that $Q_{\mathcal{L}} = \bigcup_{m \in \mathcal{M}} Q_m$ is a finite set of program positions, where Q_m is the program positions of a method m of this library; $\rightarrow_{\mathcal{L}} = \bigcup_{m \in \mathcal{M}} \rightarrow_m$ is a control-flow transition relation, where for each $m \in \mathcal{M}$, $(Q_m, PCom, \rightarrow_m, i_m, f_m)$ is a control-flow graph with a unique initial position i_m and a unique final position f_m; $InitV_{\mathcal{L}} : \mathcal{X} \rightarrow \mathcal{D}$ is an initial valuation for its memory locations.

The most general client of a library is a special client program that is used to exhibit all possible behaviors of the library. Formally, the most general client $\mathcal{MGC}$ of library $\mathcal{L}$ is defined as a tuple $(\{q_c, q'_c\}, \rightarrow_c)$, where q_c and q'_c are two program positions, $\rightarrow_c = \{(q_c, havoc, q'_c)\} \cup \{(q'_c, call(m), q_c) | m \in \mathcal{M}\}$ is a control-flow transition relation and $(\{q_c, q'_c\}, PCom, \rightarrow_c, q_c, q_c)$ is a control-flow graph. Intuitively, the most general client repeatedly calls an arbitrary method with an arbitrary argument for arbitrarily many times.

2.3 TSO operational semantics

Assume a concurrent system consists of n processes, each of which runs the most general client program of a library on a separate processor. Then, the operational semantics of a library can be defined in the context of the concurrent system.

For a library $\mathcal{L}=(Q_{\mathcal{L}}, \rightarrow_{\mathcal{L}}, InitV_{\mathcal{L}})$, its operational semantics on the TSO memory model is defined as an LTS $[\![\mathcal{L}, n]\!]_{te}$[1] $= (Conf_{te}, \Sigma_{te}, \rightarrow_{te}, InitConf_{te})$, where $Conf_{te}$, Σ_{te}, $\rightarrow_{te}$, $InitConf_{te}$ are defined as follows.

[1] "t" represents TSO memory model. "e" represents that the operational semantics in this paper extends standard TSO operational semantics [17] similarly as [8].

Each configuration of $Conf_{te}$ is a tuple (p, d, u, r, l), where

- $p : \{1, \ldots, n\} \rightarrow \{q_c, q_c'\} \cup Q_{\mathcal{L}}$ represents control states of each process;
- $d : \mathcal{X} \rightarrow \mathcal{D}$ represents values at each memory location;
- $u : \{1, \ldots, n\} \rightarrow (\{(x, a)|x \in \mathcal{X}, a \in \mathcal{D}\} \cup \{call(m, a)|m \in \mathcal{M}, a \in \mathcal{D}\} \cup \{return(m, a)|m \in \mathcal{M}, a \in \mathcal{D}\})^*$ represents contents of each processor-local store buffer; each processor-local store buffer may contain a finite sequence of pending write, pending call or pending return actions;
- $r : \{1, \ldots, n\} \rightarrow (\mathcal{R} \rightarrow \mathcal{D})$ represents values of the registers of each process.
- $l \subseteq \{1, \ldots, n\}$ contains all the processes that can currently execute commands.

Σ_{te} consists of the following subsets of actions as transition labels.

- Internal actions: $\{\tau(i)|1 \leq i \leq n\}$;
- Read actions: $\{read(i, x, a)|1 \leq i \leq n, x \in \mathcal{X}, a \in \mathcal{D}\}$;
- Write actions: $\{write(i, x, a)|1 \leq i \leq n, x \in \mathcal{X}, a \in \mathcal{D}\}$;
- Lock actions: $\{lock(i)|\ 1 \leq i \leq n\}$;
- Unlock actions: $\{unlock(i)|\ 1 \leq i \leq n\}$;
- Flush actions: $\{flush(i, x, a)|\ 1 \leq i \leq n, x \in \mathcal{X}, a \in \mathcal{D}\}$;
- Call actions: $\Sigma_{cal} = \{call(i, m, a)|1 \leq i \leq n, m \in \mathcal{M}, a \in \mathcal{D}\}$;
- Return actions: $\Sigma_{ret} = \{return(i, m, a)|1 \leq i \leq n, m \in \mathcal{M}, a \in \mathcal{D}\}$;
- Flush call actions: $\Sigma_{fcal} = \{flushCall(i, m, a)|1 \leq i \leq n, m \in \mathcal{M}, a \in \mathcal{D}\}$;
- Flush return actions: $\Sigma_{fret} = \{flushReturn(i, m, a)|1 \leq i \leq n, m \in \mathcal{M}, a \in \mathcal{D}\}$.

The initial configuration $InitConf_{te} \in Conf_{te}$ is a tuple $(p_{init}, InitV_{\mathcal{L}}, u_{init}, r_{init}, l_{init})$, where $p_{init}(i) = q_c$, $u_{init}(i) = \epsilon$ (representing an empty buffer), $r_{init}(i)(r) = regV_{init}$ (a special initial value of a register) and $l_{init} = \{1, \ldots, n\}$ for $1 \leq i \leq n, r \in \mathcal{R}$;

The transition relation $\rightarrow_{te}$ is the least relation satisfying the transition rules shown in Fig. 1. Our operational semantics is similar to the one presented in [8].

- *Register-Assign* rule: A function $f_{re} : (\mathcal{R} \rightarrow \mathcal{D}) \times \mathcal{RE} \rightarrow \mathcal{D}$ is used to evaluate register expression *re* under register valuation *rv* of current process, and its value is assigned to register r_1.
- *Library-Havoc* and $\mathcal{MGC}$*-Havoc* rules: *havoc* commands are executed for libraries and the most general clients respectively.
- *Assume* rule: If the value of register r_1 is *true*, current process can execute *assume* command. Otherwise, it must wait.
- *Read* rule: A function $lookup(u, d, i, x)$ is used to search for the latest value of x from its processor-local store buffer or the main memory, i.e.,

$$lookup(u,d,i,x) = \begin{cases} a & \text{if} \quad u(i) \uparrow_{\Sigma_x} = (x, a) \cdot l,\ \textit{for some}\ l \in \Sigma_x^* \\ d(x) & \text{otherwise} \end{cases}$$

  where $\Sigma_x = \{(x, a)|a \in \mathcal{D}\}$ is the set of pending write actions for x.
  A read action will take the latest value of x from its processor-local store buffer if possible, otherwise, it looks up the value in memory.
- *Write* rule: A write action will insert a pair of a location and a value to the tail of its processor-local store buffer.
- *Lock* and *unlock* rules: Only when the processor-local store buffer is empty, a processor can perform lock or unlock commands. A process executing lock makes itself the only active process and prevents other processes from executing commands. After a process executes unlock command, other processes become active. Thus, the commands executed from lock to unlock are not interleaved with commands of other processes.

$$\frac{i \in l, p(i) = q_1, q_1 \xrightarrow{r_1 = re}_{\mathcal{L}} q_2, r(i) = rv, f_{re}(rv, re) = a}{(p, d, u, r, l) \xrightarrow{\tau(i)}_{te} (p[i : q_2], d, u, r[i : rv[r_1 : a]], l)} \textit{Register-Assign}$$

$$\frac{i \in l, p(i) = q_1, q_1 \xrightarrow{havoc}_{\mathcal{L}} q_2, rv \in \mathcal{R} \to \mathcal{D}}{(p, d, u, r, l) \xrightarrow{\tau(i)}_{te} (p[i : q_2], d, u, r[i : rv], l)} \textit{Library-Havoc}$$

$$\frac{i \in l, p(i) = q_c, rv \in \mathcal{R} \to \mathcal{D}}{(p, d, u, r, l) \xrightarrow{\tau(i)}_{te} (p[i : q_c'], d, u, r[i : rv], l)} \mathcal{MGC}\textit{-Havoc}$$

$$\frac{i \in l, p(i) = q_1, q_1 \xrightarrow{assume(r_1)}_{\mathcal{L}} q_2, r(i)(r_1) = true}{(p, d, u, r, l) \xrightarrow{\tau(i)}_{te} (p[i : q_2], d, u, r, l)} \textit{Assume}$$

$$\frac{i \in l, p(i) = q_1, q_1 \xrightarrow{read(x, r_1)}_{\mathcal{L}} q_2, r(i) = rv, lookup(u, d, i, x) = a}{(p, d, u, r, l) \xrightarrow{read(i, x, a)}_{te} (p[i : q_2], d, u, r[i : rv[r_1 : a]], l)} \textit{Read}$$

$$\frac{p(i) = q_1, q_1 \xrightarrow{write(r_1, x)}_{\mathcal{L}} q_2, r(i)(r_1) = a, u(i) = s}{(p, d, u, r, l) \xrightarrow{write(i, x, a)}_{te} (p[i : q_2], d, u[i : (x, a) \cdot s], r, l)} \textit{Write}$$

$$\frac{p(i) = q_1, q_1 \xrightarrow{lock}_{\mathcal{L}} q_2, u(i) = \epsilon, l = \{1, \ldots, n\}}{(p, d, u, r, l) \xrightarrow{lock(i)}_{te} (p[i : q_2], d, u, r, \{i\})} \textit{Lock}$$

$$\frac{p(i) = q_1, q_1 \xrightarrow{unlock}_{\mathcal{L}} q_2, u(i) = \epsilon, l = \{i\}}{(p, d, u, r, l) \xrightarrow{unlock(i)}_{te} (p[i : q_2], d, u, r, \{1, \ldots, n\})} \textit{Unlock}$$

$$\frac{i \in l, u(i) = s \cdot (x, a)}{(p, d, u, r, l) \xrightarrow{flush(i, x, a)}_{te} (p, d[x : a], u[i : s], r, l)} \textit{Flush}$$

$$\frac{p(i) = q_c', r(i)(r_f) = a, u(i) = s}{(p, d, u, r, l) \xrightarrow{call(i, m, a)}_{te} (p[i : i_m], d, u[i : call(m, a) \cdot s], r, l)} \textit{Call}$$

$$\frac{p(i) = f_m, r(i)(r_f) = a, u(i) = s}{(p, d, u, r, l) \xrightarrow{return(i, m, a)}_{te} (p[i : q_c], d, u[i : return(m, a) \cdot s], r, l)} \textit{Return}$$

$$\frac{i \in l, u(i) = s \cdot call(m, a)}{(p, d, u, r, l) \xrightarrow{flushCall(i, m, a)}_{te} (p, d, u[i : s], r, l)} \textit{Flush-Call}$$

$$\frac{i \in l, u(i) = s \cdot return(m, a)}{(p, d, u, r, l) \xrightarrow{flushReturn(i, m, a)}_{te} (p, d, u[i : s], r, l)} \textit{Flush-Return}$$

Fig. 1 Transition rules of $\rightarrow_{te}$

- *Flush* rule: The memory system may decide to flush the entry at the head of a processor-local store buffer to memory at any time.
- *Call* and *return* rules: To deal with *call* command, a call marker is added into the tail of processor-local store buffer and current process starts to execute the initial position of method m. When the process comes to the final position of method m it can launch a *return* action, add a return marker to the tail of processor-local store buffer and start to execute the most general client.
- *Flush-Call* and *Flush-Return* rules: The call and return marker can be discarded when they are at the head of processor-local store buffer. Such actions are used to define TSO-to-TSO linearizability only.

Given above commands, a memory barrier can be implemented as "lock;unlock". The *cas* (compare-and-swap) commands can also be implemented. A *cas* command takes three

arguments: a memory location x, an expected value a and a new value b. It atomically reads x, updates it with b and returns 1 when the value of x is a; otherwise, it does nothing and return 0. As in [8], $cas(x, a, b)$ is defined as syntactic sugar for the control-flow graph representation of:

```
lock;
if (x==a) {x=b; unlock; return 1;}
{unlock; return 0;}
```

3 TSO-to-TSO linearizability and equivalent characterization

In this section we introduce the definition of TSO-to-TSO linearizability and then prove that it can be equivalently characterized by extended history inclusion. We also give definitions of variants of histories.

3.1 TSO-to-TSO linearizability

The behavior of a library is typically represented by histories of interactions between the library and the clients calling it (through call and return actions). A finite sequence $h \in (\Sigma_{cal} \cup \Sigma_{ret})^*$ is a history of an LTS $\mathcal{A}$ if there exists a trace t of $\mathcal{A}$ such that $t \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})} = h$. Let *history*($\mathcal{A}$) denote all the histories of $\mathcal{A}$.

TSO-to-TSO linearizability is a variant of linearizability on the TSO memory model. It additionally concerns the behavior of a library in the context of processor-local store buffers, i.e., the interactions between the library and store buffers through flush call and flush return actions. A finite sequence $eh \in (\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})^*$ is an extended history of an LTS $\mathcal{A}$ if there exists a trace t of $\mathcal{A}$ such that $t \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})} = eh$. Let *ehistory*($\mathcal{A}$) denote all the extended histories of $\mathcal{A}$, and $eh|_i$ the projection of eh to the actions of the ith process. Two extended histories eh_1 and eh_2 are equivalent, if for each $1 \leq i \leq n$, $eh_1|_i = eh_2|_i$.

Definition 1 (*TSO-to-TSO linearizability* [8]) For any two extended histories eh_1 and eh_2 of libraries, eh_1 is *TSO-to-TSO linearizable* to eh_2, if

- eh_1 and eh_2 are equivalent;
- there is a bijection $\pi : \{1, \ldots, |eh_1|\} \to \{1, \ldots, |eh_2|\}$ such that for any $1 \leq i \leq |eh_1|$, $eh_1(i) = eh_2(\pi(i))$;
- for any $1 \leq i < j \leq |eh_1|$, if $(eh_1(i) \in \Sigma_{ret} \cup \Sigma_{fret}) \wedge (eh_1(j) \in \Sigma_{cal} \cup \Sigma_{fcal})$, then $\pi(i) < \pi(j)$.

For two libraries $\mathcal{L}_1$ and $\mathcal{L}_2$, we say that $\mathcal{L}_2$ *TSO-to-TSO linearizes* $\mathcal{L}_1$, if for any $eh_1 \in$ *ehistory*($[\![\mathcal{L}_1, n]\!]_{te}$), there exists $eh_2 \in$ *ehistory*($[\![\mathcal{L}_2, n]\!]_{te}$), such that eh_1 is *TSO-to-TSO linearizable* to eh_2.

Informally speaking, if eh_1 is *TSO-to-TSO Linearizable* to eh_2, then eh_2 keeps all the non-overlapping pairs of call/flush call and return/flush return actions in eh_1 in the same order. It is proved in [8] that TSO-to-TSO linearizability satisfies a so-called abstraction theorem. Therefore, if $\mathcal{L}_2$ TSO-to-TSO linearizes $\mathcal{L}_1$, then it is safe to replace $\mathcal{L}_1$ with $\mathcal{L}_2$ and this will not introduce any new behaviors in the view of client programs.

The following is an example of implementation library and its specification library of TSO-to-TSO linearizability in [8]. The implementation library, spinlock, of software lock is used in various version of the Linux kernel [4] and is shown in (a):

Method release writes 1 to x without executing a memory barrier or lock, which can result in a delay for other processes on TSO. The abstraction library is given in (b) as follows:

```
word x = 1;
acquire() {
  while (1) {
    lock;
    x-;
    if (x >= 0) {
      unlock;
      return;
    }
    unlock;
    while (x <= 0) ;
  }
}

release() { x = 1; }
                 (a)
```

```
word x = 1;
acquire() {
  lock;
  assume(x == 1);
  x = 0;
  unlock;
}

release() { x = 1; }
                 (b)
```

Here, the write to x in the release of abstraction library can also be delayed in the store buffer. It can be seen that the resulting specification still guarantees mutual exclusion. It has been proved in [8] that the abstraction library TSO-to-TSO linearizes the implementation library.

Apart from histories and extended histories, there are other forms of sequences that are used to represent behaviors of libraries. For example, in [9,10] the behavior of a method starts with call action and ends with flush return action. In such situation the behavior of a library essentially contains sequences of call and flush return actions. To deal with all possible variants of histories, we generalize the notions of history as follows: Let *cal*, *ret*, *fcal* and *fret* represent call, return, flush call and flush return actions, respectively. Given distinct $x, y, z, w \in \{cal, ret, fcal, fret\}$, a (x)-history is a sequence of x actions, a (x, y)-history is a sequence of x and y actions, a (x, y, z)-history is a sequence of x, y and z actions, and a (x, y, z, w)-history is a sequence of x, y, z and w actions. It is easy to see that there are fifteen variants of histories, while the (standard) histories can be defined as $(call, ret)$-histories, and extended histories can be defined as $(call, ret, fcal, fret)$-histories.

3.2 Equivalence characterization

To handle the decision problem of TSO-to-TSO linearizability, we show that the extended history inclusion is an equivalent characterization of TSO-to-TSO linearizability. It is obvious that extended history inclusion implies TSO-to-TSO linearizability. To prove the opposite direction, we need to prove that if $\mathcal{L}_2$ TSO-to-TSO linearizes $\mathcal{L}_1$, $eh_1 \in ehistory([\![\mathcal{L}_1, n]\!]_{te})$,

$eh_2 \in ehistory([\![\mathcal{L}_2, n\ ]\!]_{te})$ and eh_1 is TSO-to-TSO linearizable to eh_2, then $eh_1 \in ehistory([\![\mathcal{L}_2, n]\!]_{te})$.

A transformation $\Rightarrow_{ER}$ is a relation between two extended histories and is defined as follows: $eh_1 \Rightarrow_{ER} eh_2$, if $eh_1 = l_1 \cdot \alpha \cdot \beta \cdot l_2$, $eh_2 = l_1 \cdot \beta \cdot \alpha \cdot l_2$ and (α, β) is neither in $(\Sigma_{ret} \cup \Sigma_{fret}) \times (\Sigma_{cal} \cup \Sigma_{fcal})$, nor actions of same process. Or we can say, eh_2 can be obtained by swapping two adjacent elements of eh_1 without violating TSO-to-TSO linearizability. We write $\Rightarrow^*_{ER}$ to denote the transition closure of $\Rightarrow_{ER}$.

Given two equivalent extended histories eh_1 and eh_2, we say that π is their bijection, if π is a bijection between $\{1, \ldots, |eh_1|\}$ and $\{1, \ldots, |eh_2|\}$, and $eh_1(i) = eh_2(\pi(i))$ for each i. We use predicate $eWit(eh_1, eh_2, i_1, i_2)$ to denote a difference between two equivalent extended histories, and $eWit(eh_1, eh_2, i_1, i_2)$ holds if $i_1 < i_2$ and $\pi^{-1}(i_1) > \pi^{-1}(i_2)$. Given two equivalent extended histories eh_1 and eh_2, a non-negative distance function $eWitSum(eh_1, eh_2)$ is used to measure the difference between them. Formally, $eWitSum(eh_1, eh_2) = |\{(m, n)|eWit(eh_1, eh_2, m, n)\ holds\}|$.

Through the well-defined transformation relation and distance function, we can show that if an extended history eh_1 is TSO-to-TSO linearizable to another extended history eh_2 and $eh_1 \neq eh_2$, then there exists a third extended history eh_3, such that

- eh_1 is TSO-to-TSO linearizable to eh_3;
- $eh_3 \Rightarrow_{ER} eh_2$;
- eh_3 is TSO-to-TSO linearizable to eh_2;
- the distance between eh_1 and eh_3 is strictly less than the one between eh_1 and eh_2.

Based on this, we prove that if eh_1 is TSO-to-TSO linearizable to eh_2, then $eh_1 \Rightarrow^*_{ER} eh_2$. Or we can say, eh_2 can be obtained from eh_1 by a finite number of $\Rightarrow_{ER}$ transformations. We also prove that if eh_2, an extended history of $[\![\mathcal{L}, n]\!]_{te}$, can be obtained from eh_1 by one step of $\Rightarrow_{ER}$ transformation, then eh_1 is also an extended history of $[\![\mathcal{L}, n]\!]_{te}$.

Based on the two results in above paragraph, it is not hard to see that TSO-to-TSO linearizability implies extended history inclusion. Therefore, extended history inclusion is an equivalent characterization of TSO-to-TSO linearizability, as presented by the following lemma.

Lemma 1 *For any two libraries $\mathcal{L}_1$ and $\mathcal{L}_2$, $\mathcal{L}_2$ TSO-to-TSO linearizes $\mathcal{L}_1$ if and only if $ehistory([\![\mathcal{L}_1, n]\!]_{te}) \subseteq ehistory([\![\mathcal{L}_2, n]\!]_{te})$.*

4 Specific libraries for classic-lossy single-channel systems

In this section, we introduce the definition of classic-lossy single-channel systems, and then show how to simulate a classic-lossy single-channel system with a concurrent library.

4.1 Classic-lossy single-channel systems

A classic-lossy single-channel system [19] is a tuple $\mathcal{S} = (Q_{cs}, \Sigma_{cs}, \{c_{cs}\}, \Gamma_{cs}, \Delta_{cs})$, where Q_{cs} is a finite set of control states, Σ_{cs} is a finite alphabet of messages, c_{cs} is the name of the single channel, Γ_{cs} is a finite set of transition labels and $\Delta_{cs} \subseteq Q_{cs} \times \Sigma^*_{cs} \times \Gamma_{cs} \times Q_{cs} \times \Sigma^*_{cs}$ is a transition relation.

Given two finite sequences $l_1 = \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_u$ and $l_2 = \beta_1 \cdot \beta_2 \cdot \ldots \cdot \beta_v$, we say that l_1 is a *subword* of l_2, denoted $l_1 \sqsubseteq l_2$, if there exists $1 \leq i_1 < \cdots < i_u \leq v$ such that for any $1 \leq j \leq u$, $\alpha_j = \beta_{i_j}$. Then, the operational semantics of $\mathcal{S}$ is given by an LTS $\mathcal{CL}(\mathcal{S})$

= ($Conf_{cs}$, Γ_{cs}, $\rightarrow_{cs}$, $initConf_{cs}$), where $Conf_{cs} = Q_{cs} \times \Sigma_{cs}^*$ is a set of configurations with $initConf_{cs} \in Conf_{cs}$ as the initial configuration. The transition relation $\rightarrow_{cs}$ is defined as follows: $(q_1, W_1) \xrightarrow{\alpha}_{cs} (q_2, W_2)$ if there exists $(q_1, U, \alpha, q_2, V) \in \Delta_{cs}$ and $W' \in \Sigma_{cs}^*$ such that $U \cdot W' \sqsubseteq W_1$ and $W_2 \sqsubseteq W' \cdot V$.

It is known that for two configurations (q_1, W_1), $(q_2, W_2) \in Conf_{cs}$ of a classic-lossy single-channel system $\mathcal{S}$, the trace inclusion between (q_1, W_1) and (q_2, W_2) is undecidable [19].

4.2 Simulation on the TSO memory model

On the TSO memory model flush actions are launched nondeterministically by the memory system. Therefore, between two consecutive $read(x, _)$ actions, more than one flush actions to x may happen. The second read action can only read the latest flush action to x, while missing the intermediate ones. These missing flush actions are similar to the missing messages that may happen in a classic-lossy single-channel system. This makes it possible to simulate a classic-lossy single-channel system with a concurrent program running on the TSO memory model. We implement such simulation through a library $\mathcal{L}_{\mathcal{S},q,W}$ specifically constructed based on a classic-lossy single-channel system $\mathcal{S}$ and a given configuration $(q, W) \in Conf_{cs}$.

For a classic-lossy single-channel system $\mathcal{S}$ =(Q_{cs}, Σ_{cs}, $\{c_{cs}\}$, Γ_{cs}, Δ_{cs}), assume the finite data domain $\mathcal{D}_{cs} = Q_{cs} \cup \Sigma_{cs} \cup \Delta_{cs} \cup \{\sharp, start, end, \bot, true, false, regV_{init}, rule_f\}$, where $Q_{cs} \cap \Sigma_{cs} = \emptyset$, $Q_{cs} \cap \Delta_{cs} = \emptyset$, $\Sigma_{cs} \cap \Delta_{cs} = \emptyset$, and the symbols $\sharp$, *start*, *end*, $\bot$, *true*, *false*, $regV_{init}$ and $rule_f$ do not exist in $Q_{cs} \cup \Sigma_{cs} \cup \Delta_{cs}$. Given a configuration $(q, W) \in Conf_{cs}$ of $\mathcal{S}$, the library $\mathcal{L}_{\mathcal{S},q,W}$ is constructed with three methods M_1, M_2 and M_3, and three private memory locations x, y and z. x is used to transmit the channel contents from M_2 to M_1, while y is used to transmit the channel contents from M_1 to M_2. z is used to transmit the transition labels of $\mathcal{CL}(\mathcal{S})$ from M_2 to M_3. It is also used to synchronize M_2 and M_3. The symbol $\sharp$ is used as the delimiter to ensure that one element will not be read twice. The symbols *start* and *end* represent the start and the end of the channel contents, respectively. $\bot$ is the initial value of x, y and z. The symbol $rule_f$ is an additional transition rule that is used to indicate the end of a simulation.

We now present the three methods in the pseudo-code, shown in Methods 1, 2 and 3. The *if* and *while* statements used in the pseudo-code can be easily implemented by the *assume* commands as well as other commands in our formation of a library. For the sake of brevity, the following macro notations are used. For sequence $l = a_1 \cdot \ldots \cdot a_m$, let *writeSeq(x,l)* denote the commands of writing $a_1, \sharp, \ldots, a_m, \sharp$ to x in sequence, and *readSeq(x,l)* denote the commands of reading $a_1, \sharp, \ldots, a_m, \sharp$ from x in sequence. We use $v := readOne(x)$ to represent the commands of reading $e, \sharp$ from x in sequence for some $e \neq \sharp$ and then assigning e to v. If $readSeq(x, l)$ or $readOne(x)$ fails to read the specified content, then the calling process will no long proceed. We use $writeOne(x, reg)$ to represent the commands of writing $a, \sharp$ to x in sequence where a is the current value of register *reg*. In the pseudo-code, r is a register in $\mathcal{R}$.

The pseudo-code of method M_1 is shown in Method 1. M_1 contains an infinite loop that never returns (Lines 1–3). At each round of the loop, it reads a new update from x and writes it to y.

Method 1: M_1

```
  Input: an arbitrary argument
1 while true do
2 |  r := readOne(x);
3 |  writeOne(y, r);
```

The pseudo-code of method M_2 is shown in Method 2. M_2 first guesses a transition rule $rule_1$, puts $rule_1$ and W into the processor-local store buffer by writing them to x (Lines 1–2). Then, it begins an infinite loop that never returns (Lines 3–16). At each round of the loop, it reads the current transition rule $rule_2 \in \Delta_{cs}$ of $\mathcal{S}$ (Line 4) and guesses a transition rule $rule_3 \in \Delta_{cs} \cup \{rule_f\}$ (Line 5). If M_2 guesses the rule $rule_f$ in Line 5, then in the next round of this loop it will be blocked at Line 4 and the simulation terminates. M_2 does not confirm $rule_2$ until it reads $start \cdot U_1$ from y at Line 6 (intermediate values of y may be lost). At Line 7, it writes $rule_3 \cdot start$ to y. Then, it reads the remaining contents of method M_1's processor-local store buffer (intermediate values of y may be lost) and writes them and V_1 to x (Lines 8–13). In Lines 14–16, it transmits the transition label α_1 to method M_3.

Method 2: M_2

```
   Input: an arbitrary argument
1  guess a transition rule rule1 = (q, _, _, _, _) ∈ Δcs ∪ {rulef};
2  writeSeq(x, rule1 · start · W · end);
3  while true do
4  |  r := readOne(y) for some rule rule2 = (q1, U1, α1, q2, V1) ∈ Δcs;
5  |  guess a transition rule rule3 that is either some (q2, _, _, _, _) ∈ Δcs or rulef;
6  |  readSeq(y, start · U1);
7  |  writeSeq(x, rule3 · start);
8  |  while true do
9  |  |  r := readOne(y);
10 |  |  if r = end then
11 |  |  |  break;
12 |  |  writeSeq(x, r);
13 |  writeSeq(x, V1 · end);
14 |  z := α1;
15 |  while z ≠ ⊥ do
16 |  |  ;
```

The pseudo-code of method M_3 is shown in Method 3. M_3 first waits for M_2 to transmit transition label to it though z by a non $\perp$ value (Lines 1–4). Then it acknowledges M_2 at Lines 5–7 and returns this transition label z at Line 8. M_3 uses lock and unlock commands to communicate with M_2. It never puts a pending write action to its processor-local store buffer. If the programming language does not provide lock and unlock commands of TSO memory model but *cas* command instead, it is safe to use $cas(z, r, \perp)$; to take place of Lines 5–7 of method M_3.

Method 3: M_3

```
  Input: an arbitrary argument
  Output: transition label for one step in CL(S)
1 while true do
2 |  r := z;
3 |  if r ≠ ⊥ then
4 |  |  break;
5 lock;
6 z := ⊥;
7 unlock;
8 return r;
```

5 Undecidability of TSO-to-TSO linearizability

As the main result of this paper, we present in this section that the TSO-to-TSO linearizability of concurrent libraries is undecidable for a bounded number of processes. We first reduce the trace inclusion problem between any two configurations of a classic-lossy single-channel system to the history inclusion problem between two specific concurrent libraries. Then, our main undecidability result follows from the equivalence between the history inclusion and the extended history inclusion for these two libraries. Recall that the latter is equivalent to TSO-to-TSO linearizability between the two libraries based on the above Lemma 1. Moreover, we prove that in general all variants of history inclusion problems, including history inclusion problem and extended history inclusion problem, are undecidable on TSO for a bounded number of processes.

5.1 Undecidability of history inclusion

In this subsection we show that given a classic-lossy single-channel system $\mathcal{S}$ and a configuration $(q, W) \in Conf_{cs}$, the histories of library $\mathcal{L}_{\mathcal{S},q,W}$ simulate exactly the paths of $\mathcal{S}$ starting from (q, W). Therefore, trace inclusion between (q_1, W_1) and (q_2, W_2) can be reduced into history inclusion between *history*$([\![\mathcal{L}_{\mathcal{S},q_1,W_1}, 3]\!]_{te})$ and *history*$([\![\mathcal{L}_{\mathcal{S},q_2,W_2}, 3]\!]_{te})$.

A path $p_{\mathcal{S}} = (q_1, W_1) \xrightarrow{\alpha_1}_{cs} (q_2, W_2) \xrightarrow{\alpha_2}_{cs} \cdots \xrightarrow{\alpha_k}_{cs} (q_{k+1}, W_{k+1}) \in path(\mathcal{CL}(\mathcal{S}), (q_1, W_1))$ is *conservative*, if the following two conditions hold: (1) it contains at least one transition, (2) assume the ith step uses rule $r_i = (q_i, U_i, \alpha_i, q_{i+1}, V_i)$ for each $1 \leq i \leq k$, then for each $1 \leq i \leq k$, there exists $W_i', W_i'' \in \Sigma_{cs}^*$ such that $U_i \cdot W_i' \sqsubseteq W_i$, $W_i'' \sqsubseteq W_i'$ and $W_{i+1} = W_i'' \cdot V_i$. Intuitively, each ith step of a conservative path does not lose any element in V_i. We prove that the traces of conservative paths equals to that of all paths for classic-lossy single-channel systems. A trace $t_{\mathcal{L}} \in trace([\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te})$ is *effective*, if $t_{\mathcal{L}}$ contains at least one return action *return*(_, M_3, _). Otherwise, it is *ineffective*.

There is actually a close connection between the conservative paths of $\mathcal{CL}(\mathcal{S})$ and the effective traces of $[\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te}$. An effective trace $t_{\mathcal{L}} \in trace([\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te})$ and a conservative path $p_{\mathcal{S}} \in path(\mathcal{CL}(\mathcal{S}), (q, W))$ *correspond*, if the sequence of return values of M_3 in $t_{\mathcal{L}}$ is the same as the sequence of transition labels of $p_{\mathcal{S}}$. The following lemma states that given a conservative path $p_{\mathcal{S}} \in path(\mathcal{CL}(\mathcal{S}), (q, W))$, there exists a corresponding effective trace $t_{\mathcal{L}} \in trace([\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te})$.

Lemma 2 *Given a conservative path $p_{\mathcal{S}} \in path(\mathcal{CL}(\mathcal{S}), (q, W))$, there exists an effective trace $t_{\mathcal{L}} \in trace([\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te})$ such that $t_{\mathcal{L}}$ and $p_{\mathcal{S}}$ correspond.*

Proof (Sketch) Assume $p_{\mathcal{S}} = (q_1, W_1) \xrightarrow{\alpha_1}_{cs} (q_2, W_2) \xrightarrow{\alpha_2}_{cs} \cdots \xrightarrow{\alpha_k}_{cs} (q_{k+1}, W_{k+1})$, where (1) $(q_1, W_1) = (q, W)$, (2) for each i, the ith transition uses rule $r_i = (q_i, U_i, \alpha_i, q_{i+1}, V_i)$, (3) for each i, $\exists W_i', W_i''$, such that $U_i \cdot W_i' \sqsubseteq W_i$, $W_i'' \sqsubseteq W_i'$ and $W_{i+1} = W_i'' \cdot V_i$.

Let us sketch how to construct a path $p_{\mathcal{L}} \in path([\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te})$ that simulates k transitions on $p_{\mathcal{S}}$. From initial configuration we first perform the following actions in sequence:

- Call M_1 in process 1 and call M_2 in process 2.
- Run M_2 from Line 1 to Line 2, write $r_1 \cdot start \cdot W_1 \cdot end$ to x while no flush action of process 2 happens during this period.

To simulate the ith $(1 \leq i \leq k)$ transition of $p_{\mathcal{S}}$, we need to perform the following actions in sequence:

- Call M_3 in process 3.
- Run M_2 from Line 3 to Line 13. M_2 reads $r_i \cdot start \cdot U_i \cdot W_i'' \cdot end$ from y and writes $r_{i+1} \cdot start \cdot W_i'' \cdot V_i \cdot end$ to x (in the case of $i = k$, M_2 write $rule_f \cdot start \cdot W_k'' \cdot V_k \cdot end$ instead). Then M_2 transmits transition label α_i to M_3 and M_3 returns α_i. Since W_{i+1} is equal to $W_i'' \cdot V_i$, M_2 writes W_{i+1} to x while it simulates the ith transition of $p_{\mathcal{S}}$.

It is obvious that $p_{\mathcal{L}}$ holds as required and its trace, $t_{\mathcal{L}}$, corresponds with $p_{\mathcal{S}}$. This completes the proof of Lemma 2. □

Figure 2 shows an example of generating a corresponding effective trace of $[\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te}$ from a conservative path of $\mathcal{CL}(\mathcal{S})$. Note that many possible executions of $[\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te}$ can get into deadlock due to the operational semantics and the pseudo-code of each method. Herein, we consider only the executions where M_3 always manages to output return labels accordingly. In Fig. 2, contents of a store buffer are written from left to right, while the time progresses from left to right, too. Assume $(q, W) = (q_1, a \cdot a)$ and there is a conservative path $p_{\mathcal{S}} = (q_1, a \cdot a) \xrightarrow{\alpha_1}_{cs} (q_2, b \cdot c) \xrightarrow{\alpha_2}_{cs} (q_3, a)$, where the first step uses rule $rule_1 = (q_1, a, \alpha_1, q_2, b \cdot c)$ (loses a in the channel), and the second one uses rule $rule_2 = (q_2, b, \alpha_2, q_3, a)$ (loses c in the channel). For this path, we can get a corresponding effective trace of $t_{\mathcal{L}}$ as follows:

1. Run M_1, M_2 and M_3 in processes P_1, P_2 and P_3 respectively. Recall that M_1 and M_2 never return, while each invocation of M_3 is associated with an interval shown in Fig. 2.
2. At Line 2 of Method 2, M_2 puts $(x, rule_1)$, $(x, \sharp)$, $(x, start)$, $(x, \sharp)$, (x, a), $(x, \sharp)$, (x, a), $(x, \sharp)$, (x, end), $(x, \sharp)$ into the store buffer of process P_2.
3. By several loops between Lines 1–3, M_1 captures the updates of x in a lossy manner, and puts $(y, rule_1)$, $(y, \sharp)$, $(y, start)$, $(y, \sharp)$, (y, a), $(y, \sharp)$, (y, end), $(y, \sharp)$ into the store buffer of process P_1.
4. At Line 4 of Method 2, M_2 captures the updates of y in a lossy manner. M_2 guesses an applicable transition rule $rule_2$, and then puts $(x, rule_2)$,$(x, \sharp)$, $(x, start)$, $(x, \sharp)$, (x, b), $(x, \sharp)$, (x, c), $(x, \sharp)$, (x, end), $(x, \sharp)$ into the store buffer of process P_2, according to transition rule $rule_1$.
5. M_2 sends the transition label α_1 to M_3 at Line 14 of Method 2. Then, M_3 returns α_1 and we finish simulating the first transition in $p_{\mathcal{S}}$.

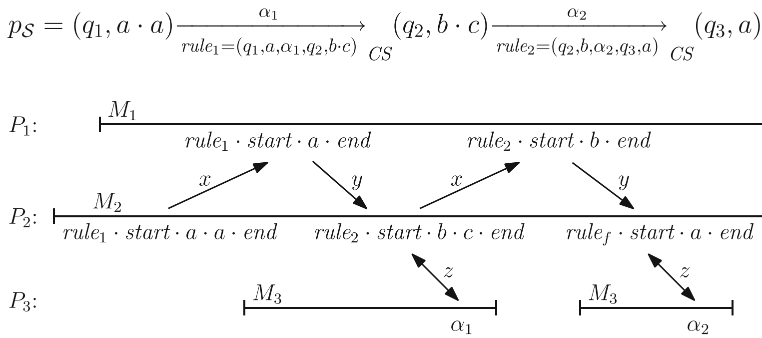

Fig. 2 A conservative path and its corresponding effective trace

6. By several loops between Lines 1–3, M_1 captures the updates of x in a lossy manner, and puts $(y, rule_2)$, $(y, \sharp)$, $(y, start)$, $(y, \sharp)$, (y, b), $(y, \sharp)$, (y, end), $(y, \sharp)$ into the store buffer of process P_1.
7. At Line 4 of Method 2, M_2 captures the updates of y. Then, M_2 decides to terminate the simulation and puts $(x, rule_f)$,$(x, \sharp)$, $(x, start)$, $(x, \sharp)$, (x, a), $(x, \sharp)$, (x, end), $(x, \sharp)$ into the store buffer of process P_2, according to transition rule $rule_2$.
8. M_2 sends the transition label α_2 to M_3 at Line 14 of Method 2. Then, M_3 returns α_2 and we finish simulating the second transition in $p_{\mathcal{S}}$.

It can be seen that $t_{\mathcal{L}}$ and $p_{\mathcal{S}}$ correspond in this example. The following lemma is the opposite direction of Lemma 2.

Lemma 3 *For each effective trace $t_{\mathcal{L}} \in trace([\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te})$, there exists a conservative path $p_{\mathcal{S}} \in path(\mathcal{CL}(\mathcal{S}), (q, W))$ such that $t_{\mathcal{L}}$ and $p_{\mathcal{S}}$ correspond.*

Proof (Sketch) Given a path $p_{\mathcal{L}} \in path([\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te})$ and let $t_{\mathcal{L}}$ be its trace. It is easy to see that the sequence of values of x which is read by M_1 is a subword of the the sequence of values of x which is written by M_2. And the sequence of values of y which is read by M_2 is a subword of the the sequence of values of y which is written by M_1.

A round of M_2 is executions from Line 1 to Line 2 or from Line 3 to Line 13 of M_2. Assume that $t_{\mathcal{L}}$ has k return actions of M_3, and for each $1 \leq i \leq k$, M_2 guesses rule $r_i = (q_i, U_i, \alpha_i, q_{i+1}, V_i)$ in its ith round. M_2 writes $r_1 \cdot start \cdot W \cdot end$ to x during its first round. Assume for each $1 < i \leq k$, M_2 reads $r_i \cdot start \cdot U_i \cdot L'_i \cdot end$ from y during its $i{+}1$th round and writes $r_i \cdot start \cdot L'_i \cdot V_i \cdot end$ to x during its $i{+}1$th round.

Recall that M_2 acts according to transition rules r_i and when M_1 reads updates of x or M_2 reads updates of y, arbitrary message can be lost. Therefore, it is not hard to see that $p_{\mathcal{S}} = (q, W) \xrightarrow{\alpha_1}_{cs} (q_2, L'_1 \cdot V_1) \xrightarrow{\alpha_2}_{cs} \cdots \xrightarrow{\alpha_k}_{cs} (q_{k+1}, L'_k \cdot V_k)$ is a conservative path of $path(\mathcal{CL}(\mathcal{S}), (q, W))$ and $t_{\mathcal{L}}$ and $p_{\mathcal{S}}$ correspond, which completes the proof of Lemma 3. □

Lemmas 2 and 3 states that there is a close connection between the conservative paths of $\mathcal{CL}(\mathcal{S})$ and the effective traces of $[\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te}$. Based on them we can now prove the following lemma, which shows that the history inclusion between concurrent libraries is undecidable on the TSO memory model for a bounded number of processes.

Lemma 4 *For any two libraries $\mathcal{L}_1$ and $\mathcal{L}_2$, it is undecidable whether $history([\![\mathcal{L}_1, 3]\!]_{te}) \subseteq history([\![\mathcal{L}_2, 3]\!]_{te})$.*

Proof (Sketch) Based on Lemmas 2 and 3, for any two configurations $(q_1, W_1), (q_2, W_2) \in Conf_{cs}$ of a classic-lossy single-channel system $\mathcal{S}$, let us prove that $history([\![\mathcal{L}_{\mathcal{S},q_1,W_1}, 3]\!]_{te}) \subseteq history([\![\mathcal{L}_{\mathcal{S},q_2,W_2}, 3]\!]_{te})$, if and only if $trace(\mathcal{CL}(\mathcal{S}), (q_1, W_1)) \subseteq trace(\mathcal{CL}(\mathcal{S}), (q_2, W_2))$.

The *only if* direction is proved by contradiction. Assume $history([\![\mathcal{L}_{\mathcal{S},q_1,W_1}, 3]\!]_{te}) \subseteq history([\![\mathcal{L}_{\mathcal{S},q_2,W_2}, 3]\!]_{te})$ but $trace(\mathcal{CL}(\mathcal{S}), (q_1, W_1))$ is not a subset of $trace(\mathcal{CL}(\mathcal{S}), (q_2, W_2))$. Thus there must exists a trace $t_{\mathcal{S}1}$, such that $t_{\mathcal{S}1} \in trace(\mathcal{CL}(\mathcal{S}), (q_1, W_1))$ and $t_{\mathcal{S}1} \notin trace(\mathcal{CL}(\mathcal{S}), (q_2, W_2))$. It is clear that $t_{\mathcal{S}1} \neq \epsilon$.

Let $p_{\mathcal{S}1}$ be the path of $t_{\mathcal{S}1}$ on $\mathcal{CL}(\mathcal{S})$ from (q_1, W_1). We can safely assume $p_{\mathcal{S}1}$ to be conservative. According to Lemma 2 there exists an effective trace $t_{\mathcal{L}1} \in trace([\![\mathcal{L}_{\mathcal{S},q_1,W_1}, 3]\!]_{te})$, such that $t_{\mathcal{L}1}$ and $p_{\mathcal{S}1}$ correspond. Let history $h = t_{\mathcal{L}1} \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})}$. It is obvious that $h \in history([\![\mathcal{L}_{\mathcal{S},q_1,W_1}, 3]\!]_{te})$ and by assumption $h \in history([\![\mathcal{L}_{\mathcal{S},q_2,W_2}, 3]\!]_{te})$.

There exists a trace $t_{\mathcal{L}2} \in trace([\![\mathcal{L}_{\mathcal{S},q_2,W_2}, 3]\!]_{te})$ such that $h = t_{\mathcal{L}2} \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})}$. It is obvious that $t_{\mathcal{L}2}$ is effective. According to Lemma 3, there exists a conservative path $p_{\mathcal{S}2} \in path(\mathcal{CL}(\mathcal{S}), (q_2, W_2))$ such that $t_{\mathcal{L}2}$ and $p_{\mathcal{S}2}$ correspond. Let trace $t_{\mathcal{S}2}$ be the trace of $p_{\mathcal{S}2}$. Thus $t_{\mathcal{S}2} \in trace(\mathcal{CL}(\mathcal{S}), (q_2, W_2))$ by its definition. Because the sequence of return values of M_3 in $t_{\mathcal{L}1}$ is same to that in $t_{\mathcal{L}2}$, $t_{\mathcal{L}1}$ and $p_{\mathcal{S}1}$ correspond, and $t_{\mathcal{L}2}$ and $p_{\mathcal{S}2}$ correspond, we can obtain that $t_{\mathcal{S}1} = t_{\mathcal{S}2}$ and $t_{\mathcal{S}1} \in trace(\mathcal{CL}(\mathcal{S}), (q_2, W_2))$, which contradicts our assumption.

The *if* direction can be similarly proved and its proof is omitted here. Therefore, the undecidability result follows from the fact that the trace inclusion problem between any two configurations of a classic-lossy single-channel system is undecidable [19]. □

5.2 Undecidability of TSO-to-TSO linearizability

Although we prove above that history inclusion is undecidable on the TSO memory model, there is still a gap between the history inclusion and the extended history inclusion between concurrent libraries. Obviously there exist libraries $\mathcal{L}_1$ and $\mathcal{L}_2$ such that $history([\![\mathcal{L}_1, n]\!]_{te}) \subseteq history([\![\mathcal{L}_2, n]\!]_{te})$ but $ehistory([\![\mathcal{L}_1, n]\!]_{te}) \nsubseteq ehistory([\![\mathcal{L}_2, n]\!]_{te})$. We show in this subsection that for the two libraries $\mathcal{L}_{\mathcal{S},q_1,W_1}$ and $\mathcal{L}_{\mathcal{S},q_2,W_2}$, corresponding to the configurations (q_1, W_1) and (q_2, W_2) of a classic-lossy single-channel system, respectively, the history inclusion and the extended history inclusion between $\mathcal{L}_{\mathcal{S},q_1,W_1}$ and $\mathcal{L}_{\mathcal{S},q_2,W_2}$ coincides on the TSO memory model.

Without loss of generality, assume M_1 and M_2 of $\mathcal{L}_{\mathcal{S},q,W}$ are called by processes P_1, P_2, respectively; while M_3 of $\mathcal{L}_{\mathcal{S},q,W}$ is repeatedly called by process P_3. Then, an extended history $eh \in ehistory([\![\mathcal{L}_{\mathcal{S},q,W}, 3]\!]_{te})$ that contains at least one return action of M_3 is in the following form:

- The first six actions of *eh* are always call and corresponding flush call actions of M_1, M_2 and M_3, while these actions may occur in any order.
- The projection of *eh* on P_i is exactly $call(i, M_i, _) \cdot flushCall(i, M_i, _)$ for $i \in \{1, 2\}$.
- Figure 3 shows the possible positions of flush call (*fcal*) and flush return (*fret*) actions in *eh*. Since M_3 always executes lock and unlock commands before it returns, during each round of a call to M_3 in P_3, the flush call action must occur before the lock action (see the dashed vertical lines in Fig. 3); hence it can only occur before the return action of M_3. During each round of a call to M_3 in P_3, the flush return action may occur alternatively at two positions: the first position is after the return action of M_3 and before the next round of a call action of M_3, as shown by the position of $fret_1$ in Fig. 3 (*a*); while the second one is after the next round of a call action of M_3 and before the consequent flush call action, as shown by the position of $fret_1$ in Fig. 3 (*b*).

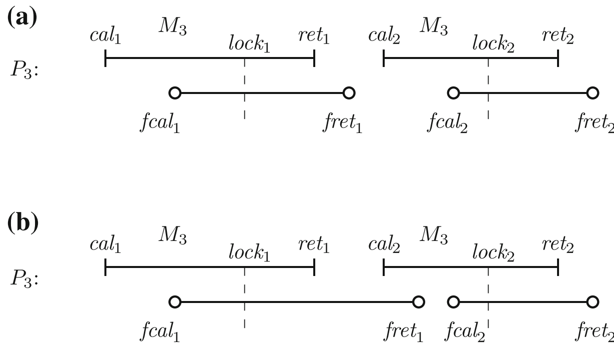

Fig. 3 Possible positions of flush call and flush return actions. **a** $fret_1$ occurs between ret_1 and cal_2, **b** $fret_1$ occurs between cal_2 and $fcal_2$

To prove that the history inclusion and the extended history inclusion coincide between libraries $\mathcal{L}_{\mathcal{S},q_1,W_1}$ and $\mathcal{L}_{\mathcal{S},q_2,W_2}$, we need to show that for an extended history eh_1 of $[\![\mathcal{L}_{\mathcal{S},q_1,W_1},3]\!]_{te}$, if eh_1 contains a return action in P_3 and $eh_1 \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})} \in history([\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te})$, then $eh_1 \in ehistory([\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te})$. Because $eh_1 \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})}$ is a history of $[\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te}$, there exists a path $p'_{\mathcal{L}}$ of $[\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te}$ corresponding to eh_1. From $p'_{\mathcal{L}}$ we can generate another path $p_{\mathcal{L}}$ of $[\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te}$ such that the extended history along $p_{\mathcal{L}}$ is exactly eh_1.

The path $p_{\mathcal{L}}$ is generated from $p'_{\mathcal{L}}$ by changing the positions of the flush return actions. Recall that during each round of a call to M_3, the flush return action may occur alternatively at two positions only. Since M_3 does not insert any pending write action into the process P_3's store buffer, $p'_{\mathcal{L}}$ can be transformed into $p_{\mathcal{L}}$ by swapping each flush return action in $p'_{\mathcal{L}}$ from its current position to the other possible one (if necessary).

An extended history is *effective* if it contains at least one $return(_, M_3, _)$ action. Otherwise, it is *ineffective*. The following lemma formalizes the idea describe above.

Lemma 5 *For a classic-lossy single-channel system* $\mathcal{S}$ *and two configurations* (q_1, W_1), $(q_2, W_2) \in Conf_{cs}$, *if* $eh_1 \in ehistory([\![\mathcal{L}_{\mathcal{S},q_1,W_1},3]\!]_{te})$ *is an effective extended history and* $eh_1 \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})} \in history([\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te})$, *then* $eh_1 \in ehistory([\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te})$.

With the help of Lemma 5, we can prove that the history inclusion and the extended history inclusion between the specific libraries coincide on the TSO memory model.

Lemma 6 *For two configurations* $(q_1, W_1), (q_2, W_2)$ *of a classic-lossy single-channel system* $\mathcal{S}$, $history([\![\mathcal{L}_{\mathcal{S},q_1,W_1},3]\!]_{te}) \subseteq history([\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te})$ *if and only if* $ehistory([\![\mathcal{L}_{\mathcal{S},q_1,W_1},3]\!]_{te}) \subseteq ehistory([\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te})$.

Proof The *if* direction is obvious.

The *only if* direction can be proved by contradiction. Assume there is an extended history eh_1 such that $eh_1 \in ehistory([\![\mathcal{L}_{\mathcal{S},q_1,W_1},3]\!]_{te})$ but $eh_1 \notin ehistory([\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te})$.

It can be seen that the sets of the ineffective extended histories of $\mathcal{L}_{\mathcal{S},q_1,W_1}$ and $\mathcal{L}_{\mathcal{S},q_2,W_2}$ are the same. By assumption, eh_1 is not an ineffective extended history of $\mathcal{L}_{\mathcal{S},q_2,W_2}$, so eh_1 must be an effective extended history of $\mathcal{L}_{\mathcal{S},q_1,W_1}$.

Let history $h = eh_1 \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})}$. It is obvious that $h \in history([\![\mathcal{L}_{\mathcal{S},q_1,W_1},3]\!]_{te})$. Then, by assumption, $h \in history([\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te})$. By Lemma 5, $eh_1 \in ehistory([\![\mathcal{L}_{\mathcal{S},q_2,W_2},3]\!]_{te})$, which contradicts the assumption. □

The undecidability of TSO-to-TSO linearizability for a bounded number of processes is a direct consequence of Lemmas 1, 4 and 6.

Theorem 1 *For any two concurrent libraries $\mathcal{L}_1$ and $\mathcal{L}_2$, it is undecidable whether $\mathcal{L}_2$ TSO-to-TSO linearizes $\mathcal{L}_1$ for a bounded number of processes.*

5.3 Undecidability of all variants of history inclusion problems

In this subsection we show that all variants of history inclusion problems are undecidable on TSO for a bounded number of processes.

By constructing a close connection between the conservative paths of $\mathcal{CL}(\mathcal{S})$ and return sequences, as in the undecidability proof of Lemma 4, it is not hard to prove that (*ret*)-history inclusion problem is also undecidable on TSO for a bounded number of processes.

Recall that flush call action has fixed positions, flush return action has two possible positions and can be swapped from one position to another position if necessary. Therefore, it can be similarly proved as in Theorem 1 that, the (*fcal*, *ret*)/ (*cal*, *fcal*, *ret*)/ (*fret*)/ (*cal*, *fret*)/ (*fcal*, *fret*)/ (*cal*, *fcal*, *fret*)/ (*ret*, *fret*)/ (*cal*, *ret*, *fret*)/ (*fcal*, *ret*, *fret*)-history inclusion problems are all undecidable on TSO for a bounded number of processes.

To prove that other variants of history inclusion problems to be undecidable, we slightly modify the library $\mathcal{L}_{\mathcal{S},q,W}$ into library $\mathcal{L}^c_{\mathcal{S},q,W}$. The only difference between $\mathcal{L}^c_{\mathcal{S},q,W}$ and $\mathcal{L}_{\mathcal{S},q,W}$ is method M_3, and the pseudo-code of method M_3 of $\mathcal{L}^c_{\mathcal{S},q,W}$ is shown in Method 4. M_3 of $\mathcal{L}^c_{\mathcal{S},q,W}$ needs to ensure that if M_3 returns, then its argument and return value must equal to one step of transitions of $\mathcal{S}$.

Method 4: M_3

Input: some value $a \in \Gamma_{cs}$
Output: transition label for one step in $\mathcal{CL}(\mathcal{S})$

```
1  while true do
2  |  r := z;
3  |  if r ≠ ⊥ then
4  |  |  break;
5  lock;
6  z := ⊥;
7  unlock;
8  while r ≠ a do
9  |  ;
10 return r;
```

According to the construction of $\mathcal{L}^c_{\mathcal{S},q,W}$, it is obvious that if $\alpha_1 \cdot \ldots \cdot \alpha_k$ is a trace of $\mathcal{CL}(\mathcal{S})$ from (q, W), then there must be a trace $t \in trace([\![\mathcal{L}^c_{\mathcal{S},q,W}, 3]\!]_{te})$, such that t contains $k+1$ call actions of M_3, and the first k arguments of M_3 are $\alpha_1, \ldots, \alpha_k$, while the last argument of M_3 is irrelevant. It is not hard to construct a close connection between the conservative paths of $\mathcal{CL}(\mathcal{S})$ and sequences of call actions (except the last one), and this also holds for flush call actions. Therefore, it is not hard to prove that, the (*cal*)/ (*fcal*)/ (*cal*, *fcal*)-history inclusion problems are undecidable on TSO for a bounded number of processes.

The following theorem states that all variants of history inclusion problems are undecidable on TSO for a bounded number of processes.

Theorem 2 *On TSO memory model, all variants of history inclusion problem are undecidable for a bounded number of processes.*

Similarly, method M_3 of $\mathcal{L}^c_{\mathcal{S},q,W}$ can use *cas* commands instead of lock and unlock commands. Therefore, on other relaxed memory models that are weaker than TSO and also provides TSO behaviors for write, read and *cas* commands , the (*cal*)/ (*ret*)/ (*cal*, *ret*)-history inclusion problems are still undecidable for a bounded number of processes.

6 Conclusion and future work

We have shown that the decision problem of TSO-to-TSO linearizability is undecidable for a bounded number of processes. The proof method is essentially by a reduction from a known undecidable problem, the trace inclusion problem of a classic-lossy single-channel system. To facilitate such a reduction, we introduced an intermediate notion of history inclusion between concurrent libraries on the TSO memory model. We then demonstrated that a configuration (q, W) of a classic-lossy single-channel system $\mathcal{S}$ can be simulated by a specific library $\mathcal{L}_{\mathcal{S},q,W}$, interacting with three specific processes on the TSO memory model. Although history inclusion does not coincide with extended history inclusion in general, they do coincide on a restricted class of libraries. We prove that $\mathcal{L}_{\mathcal{S},q,W}$ lies within such class. Finally, our undecidability result follows from the equivalence between extended history inclusion and TSO-to-TSO linearizability.

The problem of the linearizability between libraries on the SC memory model [11] can be shown to be decidable for a bounded number of processes. This is due to the provable equivalence between history inclusion and linearizability on the SC memory model, while the former is decidable. Thus, our work states clearly a boundary of decidability for linearizability of concurrent libraries on various memory models. As by-product of this work, we prove that all variants of history inclusion problems are undecidable on TSO for a bounded number of processes. This reveals that the undecidability of TSO-to-TSO linearizability comes from the unbounded size of processor-local store buffer, instead of which actions are chosen.

Other relaxed memory models, such as the memory models of POWER and ARM, are much weaker than the TSO memory model. We conjecture that variants of linearizability on these relaxed memory models may also be reduced to some new forms of extended history inclusion, similar to the variants of linearizability for C/C++ memory model in [3], and these variants should also be undecidable. However, the decision problem of TSO-to-SC linearizability, which amounts to checking whether histories of a library on the TSO memory model belong to a regular language, still remains open. For concurrent programs using write, read and *cas* commands but not call and return actions, Atig et al. proved in [2] that the reachability problem between any two configurations is decidable for a bounded number of processes on the TSO memory model. However, this reachability problem turns much more complex when call and return actions are involved. In [21], we have already proved that if the number of call and return actions is bounded in a history, the decision problem of TSO-to-SC linearizability is decidable for a bounded number of processes. As future work, we would like to further investigate the decidability of TSO-to-SC linearizability and other variants of linearizability for relaxed memory models.

References

1. Alur, R., McMillan, K., Peled, D.: Model-checking of correctness conditions for concurrent objects. In: LICS 1996, pp. 219–228. IEEE Computer Society (1996)
2. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL 2010, pp. 7–18. ACM (2010)
3. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: Giacobazzi, R., Cousot, R. (eds.) POPL 2013, pp. 235–248. ACM (2013)
4. Bovet, D., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly, Sebastopol (2005)
5. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) POPL 2011, pp. 55–66. ACM (2011)
6. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013, pp. 290–309. Springer (2013)
7. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: Rajamani, S.K., Walker, D. (eds.) POPL 2015, pp. 651–662. ACM (2015)
8. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model. In: Seidl, H. (eds.) ESOP 2012, pp. 87–107. Springer (2012)
9. Derrick, J., Smith, G., Groves, L., Dongol, B.: Using coarse-grained abstractions to verify linearizability on TSO. In: Yahav, E. (eds.) HVC 2014, pp. 1–16. Springer (2014)
10. Derrick, J., Smith, G., Dongol, B.: Verifying linearizability on TSO architectures. In: Albert, E., Sekerinski, E. (eds.) IFM 2014, pp. 341–356. Springer (2014)
11. Filipovic, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. In: Castagna, G. (eds.) ESOP 2009, pp. 252–266. Springer (2009)
12. Gotsman, A., Musuvathi, M., Yang, H.: Show no weakness: sequentially consistent specifications of TSO libraries. In: Aguilera, M.K. (eds.) DISC 2012, pp. 31–45. Springer (2012)
13. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
14. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess program. IEEE Trans. Comput. **28**(9), 690–691 (1979)
15. Liu, Y., Chen, W., Liu, Y.A., Sun, J., Zhang, S.J., Dong, J.S.: Verifying linearizability via optimized refinement checking. IEEE Trans. Softw. Eng. **39**(7), 1018–1039 (2013)
16. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: Palsberg, J., Abadi, M. (eds.) POPL 2005, pp. 378–391. ACM (2005)
17. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009, pp. 391–407. Springer (2009)
18. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Hall, M. W., Padua, D.A. (eds.) PLDI 2011, pp. 175–186. ACM (2011)
19. Schnoebelen, P.: Bisimulation and other undecidable equivalences for lossy channel systems. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001, pp. 385–399. Springer (2001)
20. Vechev, M.T., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Pasareanu, C.S. (ed.) SPIN 2009, pp. 261–278. Springer (2009)
21. Wang, C., Lv, Y., Wu, P.: Bounded TSO-to-SC linearizability is decidable. In: Freivalds, M.R., Engels, G., Catania, B. (eds.) SOFSEM 2016, pp. 404–417. Springer (2016)
22. Wang, C., Lv, Y., Wu, P.: TSO-to-TSO linearizability is undecidable. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015, pp. 309–325. Springer (2015)