Compositional reasoning about concurrent libraries on the axiomatic TSO memory model

Artem Khyzha
IMDEA Software Institute
artem.khyzha@imdea.org

Alexey Gotsman
IMDEA Software Institute
alexey.gotsman@imdea.org

Abstract

Linearizability is a commonly accepted notion of correctness for libraries of concurrent algorithms, which has recently started to become adopted for weaker consistency guarantees provided by hardware and software platform. In this paper, we present the first definition of linearizability on the axiomatically formulated Total Store Order weak memory model, implemented by x86 processors. We establish that our definition is a correct one in the following sense: while proving a property of a client of a concurrent library, we can soundly replace the library by its abstract implementation related to the original one by our generalisation of linearizability. This allows abstracting from the details of the library implementation while reasoning about the client.

1 Introduction

Modern multiprocessor architectures, such as Intel x86 [5], IBM POWER [6, 4] and ARM, provide memory consistency models that are *weaker* than the classical sequential consistency (SC). What makes these models different is that they do certain relaxations to the order of memory accesses, which make program execution not sequentially consistent. Relying on relaxations enables implementing programs more efficiently, but leads to counter-intuitive behaviours in many cases, so programming on weak memory models can be subtle and error-prone.

Compositional reasoning about programs on the weak memory models requires a new formalisation for **correctness** of program components. Correctness of concurrent libraries is commonly formalised by the notion of *linearizability* [3], which fixes a certain correspondence between the library and its (usually sequential) abstract specification with methods implemented atomically. Unfortunately, the classical definition of linearizability is only appropriate for sequentially consistent (SC) memory models, in which accesses to shared memory occur in a global-time linear order.

In this paper we suggest an approach for compositional reasoning on a weak memory model of Total Store Order (TSO), implemented by x86 processors [5] (Sections 2, 3). TSO allows the store buffer optimisation implemented by modern multiprocessors: writes performed by a processor are buffered in a processor-local store buffer and are flushed into the memory at some later time.

A consequence of the store buffer optimisation is that on TSO, given two memory locations x and y initially holding 0, if two CPUs respectively write 1 to x and y and then read from y and x, as in the following program, it is possible for both to read 0 in the same execution:

$$\begin{array}{c} \{x = y = 0\} \\ x = 1; \quad \| \quad y = 1; \\ b = y; \quad \| \quad a = x; \\ \{a = b = 0\} \end{array}$$

This happens when the reads from y and x occur before the writes to them have propagated from the store buffers of the corresponding CPUs to the memory. To exclude such behaviours, TSO processors provide special instructions, called *memory fences*, that force the store buffer of the corresponding CPU

to be flushed completely before executing the next instruction. Adding memory fences after the writes to x and y in the above program would make it produce only SC behaviours.

In this paper, we present the definition of linearizability on a weak memory model of TSO, which is different from a classic definition due to the store buffer relaxation. Usually the semantics of weak memory models is described in operational or axiomatic setting, and in this work we choose the axiomatic way. While operational model is more intuitive, the axiomatic semantics is more abstracted from a particular implementation and in some situations is easier to reason about.

We show that our definition of linearizability is a right one in the sense that it validates what we call the Abstraction Theorem (Theorem 4, Section 4): while proving a property of a client of a concurrent library, we can soundly replace the library by its abstract implementation related to the original one by our generalisation of linearizability. Abstraction theorem has a practical value as a compositional reasoning and verification technique: it enables abstracting from the details of the library implementation while reasoning about its client, despite subtle interactions between the two caused by the weak memory model.

2 Preliminaries

The most intuitive way to explain TSO is to define its operational semantics using an abstract machine. In the following, we informally present the operational model, previously described in [5], and then in Section 3 we formally define axiomatic semantics. Due to space constraints we do not provide a proof of their equivalence.

Programming language. We assume that the memory consists of locations $\mathsf{Loc} = \{1, 2, \ldots\}$ containing values $\mathsf{Val} = \mathbb{Z}$. We consider programs in the following core language:

$$C ::= \alpha \mid C; C \mid C + C \mid C^* \mid m \qquad L ::= \{m = C_m \mid m \in M\} \qquad C(L) ::= \mathsf{let}\ L\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n$$

A program consists of a ***library*** L implementing methods $m \in \mathsf{Method}$ and its ***client*** $C_1 \parallel \ldots \parallel C_n$, given by a parallel composition of threads (for simplicity, in this paper we suppose that all threads are bijectively mapped to a set of CPUs). Threads are indexed by $\mathsf{ThreadID} = \{1, \ldots, n\}$. The commands include primitive commands $\alpha \in \mathsf{PComm}$, method calls $m \in \mathsf{Method}$, sequential composition $C; C'$, non-deterministic choice $C + C'$ and iteration C^*. We use $+$ and * instead of conditionals and while loops for theoretical simplicity: given appropriate primitive commands, the latter can be defined in the language as syntactic sugar.

We assume that every method accepts a single parameter and returns a single value. Parameters and return values are passed by every thread via distinguished locations in memory, denoted $\mathsf{param}_t, \mathsf{retval}_t \in \mathsf{Loc}$ for $t \in \mathsf{ThreadID}$. The rest of memory locations are partitioned into those owned by the client (CLoc) and the library (LLoc): $\mathsf{Loc} = \mathsf{CLoc} \uplus \mathsf{LLoc} \uplus \{\mathsf{param}_t, \mathsf{retval}_t \mid t \in \mathsf{ThreadID}\}$.

TSO operational semantics. In the operational semantics we consider an abstract machine executing programs in the core language. Each CPU has a set of general-purpose registers $\mathsf{Reg} = \{\mathtt{r}_1, \ldots, \mathtt{r}_m\}$ storing values from Val. On TSO, processors do not write to memory directly. Instead, every CPU has a ***store buffer***, which holds write requests that were issued by the CPU, but have not yet been ***flushed*** into the shared memory. The state of a buffer is described by a sequence of location-value pairs.

The abstract machine can perform the following transitions:

- A CPU wishing to write a value to a memory location adds an appropriate entry to the *tail* of its store buffer.
- The entry at the *head* of the store buffer of a CPU is flushed into the memory at a non-deterministically chosen time. Store buffers thus have the FIFO ordering.
- A CPU can execute a *memory fence* that flushes all the content of its store buffer to the memory in the FIFO ordering.

- $\langle \mathsf{skip} \rangle = \{(\emptyset, \emptyset)\}$;
- $\langle C_1; C_2 \rangle_t = \{(A_1 \uplus A_2, \mathsf{po}_1 \cup \mathsf{po}_2 \cup \{(a,b) \mid a \in A_1 \wedge b \in A_2\})\}$;
- $\langle C_1 + C_2 \rangle_t = \langle C_1 \rangle_t \cup \langle C_2 \rangle_t$;
- $\langle C^* \rangle_t = \{\emptyset, \emptyset\} \cup \{(\biguplus_{i=1}^n A_i, \bigcup_{i=1}^n \mathsf{po}_i \cup \{(a,b) \mid a \in A_i \wedge b \in A_j \wedge i < j\}) \mid (A_i, \mathsf{po}_i) \in \langle C \rangle_t \wedge n \geq 1\}$;
- $\langle m \rangle_t = \{(A \uplus \{c\} \uplus \{d\}, \mathsf{po} \cup \{(c,d)\} \cup \{(c,a),(a,d) \mid a \in A\}) \mid (A, \mathsf{po}) \in \langle C_m \rangle_t \wedge c = (\_, t, \mathsf{call}\ m(\_)) \wedge d = (\_, t, \mathsf{ret}\ m(\_))\}$;
- $\langle \mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n \rangle = \mathsf{prefix}(\{(\biguplus_{t=1}^n A_t, \bigcup_{t=1}^n \mathsf{po}_t) \mid (A_t, \mathsf{po}_t) \in \langle C_t \rangle_t,\ t = 1..n\})$, where $\mathsf{prefix}(A, \mathsf{po}) = \bigcup_{a \in A} (\{a' \mid a' \in A \wedge a' \xrightarrow{\mathsf{po}} a\}, \mathsf{po}) \cup \{(A, \mathsf{po})\}$.

Figure 1: Program order semantics of common commands

- A CPU wishing to read from a memory location first looks it up in its store buffer. If there are entries for this location, it reads the value from the newest one; otherwise, it reads the value directly from the memory.
- A CPU can execute a command affecting only its registers. In particular, it can call a library method or return from it.

3 The TSO axiomatic memory model

Action structures. We record information about program executions using ***actions***, defined as follows:

$$a \in \mathsf{Act} ::= (e,t,\mathsf{store}(x,v)) \mid (e,t,\mathsf{load}(x,v)) \mid (e,t,\mathsf{call}\ m(v)) \mid (e,t,\mathsf{ret}\ m(v)) \mid (e,t,\mathsf{fence})$$

Here $t \in \mathsf{ThreadID}$, $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, and e is an *action identifier*, picked from the set AId. For call and return actions v means actual parameter and return value respectively. We omit e annotation from actions, when it is not relevant, and often use r, w and f to denote load, store and fence actions.

We denote the set of all finite sets of actions with $\mathscr{P}(\mathsf{Act})$. When considering a relation R over actions, we write $(a,b) \in \mathsf{R}$ and $a \xrightarrow{\mathsf{R}} b$ interchangeably.

Program order semantics. We first define a *program order semantics*, which generates all executions of a program based solely on the structure of its statements, without taking into account the semantics of memory operations. For instance, in generated executions loads can read arbitrary values disregarding the values written by performed store actions. After we define the notion of execution, we introduce axioms which filter out executions that do not satisfy them.

Program order semantics associates a program with a set of ***action structures***—tuples (A, po), where $A \in \mathscr{P}(\mathsf{Act})$, and $\mathsf{po} : A \times A$ is ***program order*** that is a total on the set of actions by the same thread.

Let $A \uplus B$ be the union of the sets of actions A and B with disjoint sets of action identifiers. Consider a program $C(L) = (\mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n)$. We define the set of action structures $\langle C(L) \rangle$ for program executions in Figure 1. By construction and definition of $\mathsf{prefix}(A, \mathsf{po})$, $\langle C(L) \rangle$ is also prefix-closed, i.e. it includes incomplete executions.

For a primitive command $\alpha \in \mathsf{PComm}$, we assume a set $\langle \alpha \rangle_t$ of all action structures α produces when executed by thread t. We require that structures in $\langle \alpha \rangle_t$ do not contain call or return actions.

The memory model gives a semantics to a program as a set of ***executions***, each of which is a tuple $X = (A, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc}, \mathsf{hb})$ of a set of actions $A \in \mathscr{P}(\mathsf{Act})$ and relations on A. An execution enriches an action structure with information about the way operations on memory are performed. The relations included into an execution are as follows:

MOWF. mo is total, transitive, irreflexive and relates only store actions in the execution.

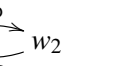
POWF. po is total, transitive, irreflexive and relates only actions by the same thread.

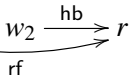
SCWF. sc is total, transitive, irreflexive and relates only fences in the execution.

RFWF. $(\forall w_1, w_2, r.\, w_1 \xrightarrow{\mathsf{rf}} r \wedge w_2 \xrightarrow{\mathsf{rf}} r \implies w_1 = w_2) \wedge$
$(\forall w, r.\, w \xrightarrow{\mathsf{rf}} r \implies \exists x, a.\, w = (\_, \mathsf{store}(x,a)) \wedge (r = (\_, \mathsf{load}(x,a))) \vee r = (\_, \mathsf{call}\ \_(a))) \vee r = (\_, \mathsf{ret}\ \_(a))))$

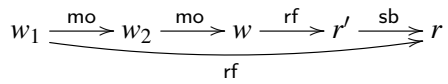
RFDET. $\forall x, w, r.(r = (\_, \mathsf{load}(x, \_)) \vee (\exists t.x = \mathsf{param}_t \wedge r = (t, \mathsf{call}\ \_)) \vee$
$(\exists t.x = \mathsf{retval}_t \wedge r = (t, \mathsf{ret}\ \_))) \wedge w = (\_, \mathsf{store}(x, \_)) \wedge (w \xrightarrow{\mathsf{hb}} r \vee w \xrightarrow{\mathsf{po}} r) \implies \exists w'.w' \xrightarrow{\mathsf{rf}} r$

HBDEF. $\mathsf{hb} = (\mathsf{po} \cup \mathsf{rf})^+$

HBVSMO. $\neg\exists w_1, w_2.$ 

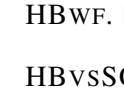
RFMR. $\neg\exists w_1, w_2, r.$ 

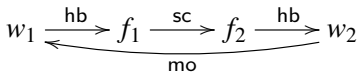
where w_1, w_2 and r access the same location.

RFMR'. $\neg\exists w, w_1, w_2, r.$


where w_1 and w_2 write to the same location, and w and r' are by different threads.

HBWF. hb is acyclic.

HBVSSC. $\neg\exists f_1, f_2.$ 

MOVSSC. $\neg\exists w_1, w_2, f_1, f_2.$


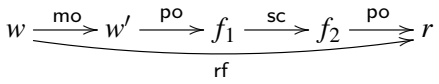
SCRF. $\neg\exists w, w', f_1, f_2, r.$


Figure 2: The validity axioms

- rf: ***reads-from***, relating the load actions r to the store actions w from which they take their values;
- mo: ***modification order***, relates all store actions in the order they hit the memory;
- sc: ***synchronisation order***, relates all fences in the order of their execution;
- hb: ***happens-before***, showing the precedence of actions in the execution.

Validity. For an execution X and one of the relations R defined above, we write $\mathsf{R}(X)$ to select the corresponding relation for X. An execution $X = (A, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc}, \mathsf{hb})$ is called ***valid*** when it satisfies the validity axioms in Figure 2.

A call action $(t, \mathsf{call}\ m(v))$ gets its argument v by reading from a correspondent client's store to param_t. A return action $(t, \mathsf{ret}\ m(v))$ gets its return value v by reading from a correspondent library's store to retval_t. In the following we treat calls and returns as loads, what is established in RFWF axiom. We also assume that call (return) actions access $param_t$ ($retval_t$) only when executed by a thread t.

A store action $(t, \mathsf{store}(x,v))$ writes a value v to the address in memory. Analogously to the operational semantics, where a written value does not hit the main memory immediately, in our model a store $(t, \mathsf{store}(x,v))$ becomes observable from a thread t, but not always from the other ones. The explicit way to make the written value visible to the other threads is to execute a fence action (t, fence). We add RFMR, RFMR' and SCRF axioms to ensure that a load action reads the most recent value that is observable to its thread.

We further use $[\![C(L)]\!]$ to denote for a given program $C(L)$ the set of all valid executions with action structures from $\langle C(L)\rangle$: $[\![C(L)]\!] = \{X \mid X = (A, \mathsf{po}, \_, \_, \_) \wedge (A, \mathsf{po}) \in \langle C(L)\rangle\}$.

Execution projections. We call actions of the form $(t, \mathsf{call}\ m(v))$ or $(t, \mathsf{ret}\ m(v))$ ***interface actions***.

Consider an execution $X = (A, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc}, \mathsf{hb})$ of $C(L)$. An action $a \in A$ is a ***library action***, if it is an interface action, or $\exists b.\ b = (\_, \mathsf{call}\ \_) \wedge b \xrightarrow{\mathsf{po}} a \wedge \neg\exists c.\, c = (\_, \mathsf{ret}\ \_) \wedge b \xrightarrow{\mathsf{po}} c \xrightarrow{\mathsf{po}} a$.

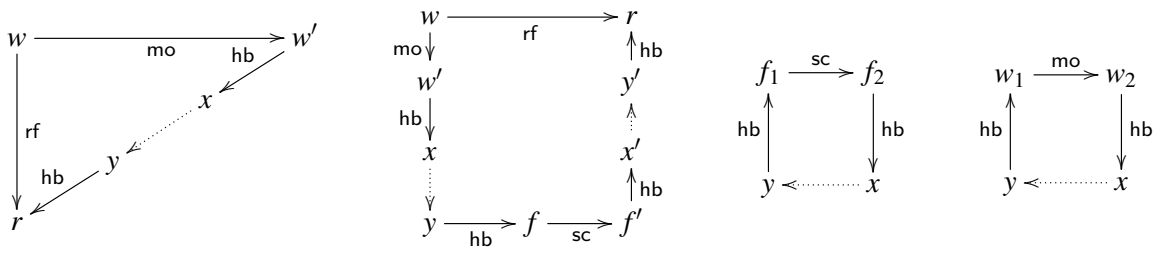


Figure 3: The definition of the $\mathsf{deny}(X)$. Here x, y, x', y' are interface actions. If the solid edges belong to the corresponding relations in X, the dashed edges belong to $\mathsf{deny}(X)$.

An action $a \in A$ is a ***client action***, if it is an interface action, or the negation of the above property holds. Let $\mathsf{client}(A)$ be the set of client actions in A. We define a client's execution:

$$\mathsf{client}(X) = (\mathsf{client}(A), \mathsf{client}(\mathsf{po}), \mathsf{client}(\mathsf{rf}), \mathsf{client}(\mathsf{mo}), \mathsf{client}(\mathsf{hb}))$$

by projecting all the relations in X to actions from $\mathsf{client}(A)$. We also use analogous projection $\mathsf{lib}(X)$ to library actions and lift client and lib to sets of executions pointwise.

Non-interference. We assume that the set of memory locations Loc is partitioned into those owned by the client (CLoc) and the library (LLoc): $\mathsf{Loc} = \mathsf{CLoc} \uplus \mathsf{LLoc}$. The client C and the library L are *non-interfering* in $C(L)$, if in every computation from $[\![C(L)]\!]$, commands performed by the client (library) code access only locations from CLoc (LLoc). Formally, an execution $C(L)$ is called ***non-interfering*** when it satisfies the following axiom:

$$\begin{aligned}&\textsc{NonInterf}.\forall a, x, t.\, a \in A \wedge (a = (t, \mathsf{store}(x, \_)) \vee a = (t, \mathsf{load}(x, \_))) \implies \\ &\quad ((a \in \mathsf{lib}(A, \mathsf{po}) \iff (x \in \mathsf{LLoc} \vee a = (t, \mathsf{store}(\mathsf{retval}_t, \_)) \vee a = (t, \mathsf{load}(\mathsf{param}_t, \_)))) \wedge \\ &\qquad (a \in \mathsf{client}(A, \mathsf{po}) \iff (x \in \mathsf{CLoc} \vee a = (t, \mathsf{store}(\mathsf{param}_t, \_)) \vee a = (t, \mathsf{load}(\mathsf{retval}_t, \_))))).\end{aligned}$$

In the following, we assume that at the beginning of execution all locations are arbitrarily and explicitly initialised by means of store actions.

4 Abstraction theorem

The idea behind linearizability is to record all interactions between the client and the library. That is done by means of the notion of a ***history***. Clients and libraries can affect each other by passing different values through interface actions. Precisely, the library can observe the parameters provided by the client at calls, and the client can observe the library's return values. Therefore, a history includes the set of interface actions in the execution. We also include in histories two partial orders (*guarantee* and *deny*) over interface actions to consider additional interactions caused by relaxations of TSO.

Definition 1. A **history** is a set of interface actions and a pair of partial orders over it.

Informally, in a history $H = (I, G, D)$, the ***guarantee*** G describes the happens-before edges enforced by the library; and the ***deny*** D describes the happens-before edges that a client must not enforce.

Consider an execution X and its interface actions $I(X)$. We let $\mathsf{guar}(X)$ be the projection of $\mathsf{hb}(X)$ onto $I(X)$ and $\mathsf{deny}(X)$ be the relation over $I(X)$ obtained from dashed edges in Figure 3. These edges describe all the possible ways in which happens-before edges enforced by the client can contradict other relations from the library execution; the axioms that can be violated are indicated in the figure. Let $\mathsf{history}(X) = (I(X), \mathsf{guar}(X), \mathsf{deny}(X))$. We lift history to sets of executions pointwise.

We use the $\subseteq$ relation between partial orders to denote that a partial order is a sub-relation of another one, and lift it to histories as follows: $(I_1, G_1, D_1) \subseteq (I_2, G_2, D_2) = (I_1 = I_2) \wedge (G_1 \subseteq G_2) \wedge (D_1 \subseteq D_2)$.

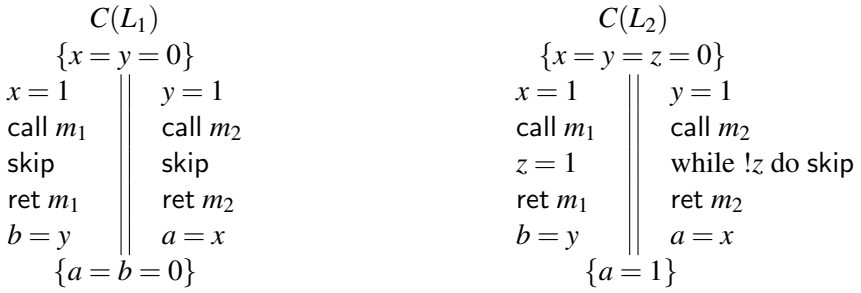


Figure 4: A motivation to include guarantee: it should not be the case that $L_1 \sqsubseteq L_2$, if the latter adds synchronisation and therefore allows less behaviours.

Definition 2. A history H' **linearizes** a history H if $H' \subseteq H$.

Thus, a linearized history enforces fewer dependencies between interface actions for a client and has less restrictions on enforcing dependencies by a client. This indeed allows more client behaviours.

To generate the set of all histories of a given library L, we consider its *most general client*, whose threads repeatedly invoke library methods in any order and with any parameters possible. Take $n \geq 1$ and assume $\mathsf{sig}(L) = \{m_1, \ldots, m_l\}$. Then we define $\mathsf{MGC}_n(L) = (\mathsf{let}\ L\ \mathsf{in}\ C_1^{\mathsf{mgc}} \parallel \ldots \parallel C_n^{\mathsf{mgc}})$, where for all t, $C_t^{\mathsf{mgc}} = (m_1 + \ldots + m_l)^*$.

A ***library execution*** of L is an execution of $[\![\mathsf{MGC}_n(L)]\!]$ for some $n \geq 1$. A library execution is ***valid***, if it satisfies the validity axioms in Figure 2, and it is ***non-interfering***, if it satisfies NONINTERF axiom.

Let $[\![L]\!]$ be the set of all valid library executions of L. The set of executions in $[\![L]\!]$ defines a *library-local semantics* of L. We say that a library L is non-interfering if so is every execution in $[\![L]\!]$.

Now we present our main result – the definition of linearizability for the axiomatic model of TSO. The correctness of the proposed notion is established in Theorem 4.

Definition 3. For non-interfering libraries L_1 and L_2, L_2 **linearizes** L_1, written $L_1 \sqsubseteq L_2$, if:

$$\forall H_1 \in \mathsf{history}([\![L_1]\!]).\, \exists H_2 \in \mathsf{history}([\![L_2]\!]).\, H_2 \subseteq H_1.$$

Noteworthily, checking linearizability $L_1 \sqsubseteq L_2$ does not involve reasoning about any client. What we need to do is to generate all possible library-local executions, or, in other words, all possible executions of $\mathsf{MGC}(L_1)$ and $\mathsf{MGC}(L_2)$, and check the definition.

Theorem 4 (Abstraction)**.** *If L_1, L_2 and $C(L_2)$ are non-interfering and $L_1 \sqsubseteq L_2$, then $C(L_1)$ is non-interfering and $\forall X_1, X_2\ .\ X_1 \in \mathsf{client}([\![C(L_1)]\!]) \wedge X_2 \in \mathsf{client}([\![C(L_2)]\!]) \wedge \mathsf{hb}(X_2) \subseteq \mathsf{hb}(X_1)$.*

By Theorem 4, while reasoning about a client $C(L_1)$ of a library L_1, we can soundly replace L_1 with a simpler library L_2 linearizing L_1: if a property over client actions holds over $C(L_2)$, it will also hold over $C(L_1)$. Since L_2 is usually simpler than L_1, this eases the proof of the resulting program. Thus, the proposed notion of linearizability and Theorem 4 enable compositional reasoning about programs on TSO: they allow decomposing the verification of a whole program into the verification of its constituent components.

Let us now return to definition of a history and illustrate the ideas behind inclusion of two relations into it. The inclusion of a *guarantee* relation into a history is aimed to prevent a library L_2 that linearizes L_1 from adding new synchronisations, so that a client cannot notice a difference between them.

Consider libraries L_1 and L_2 with different implementations for methods m_1 and m_2 along with their client C in Figure 4. A client is able to observe a synchronisation inside a library L_2, because it disables some client's behaviours. Particularly, because of reading z in m_2, a store $x = 1$ is always flushed by the moment of reading $a = x$. Consequently, the outcome $b = 0$ is possible in $C(L_1)$ and never happens in $C(L_2)$. In our setting such case is ruled out by a guarantee relation, since all histories of L_2 contain an edge (between a call to m_1 and a return from m_2) that any history of L_1 does not.

Figure 5: A motivation to include deny: if $L_1 \sqsubseteq L_2$, then client must not be able to distinguish them by making a synchronisation between library methods calls.

With the following example we show the role of a *deny* relation in a history. Consider libraries L_1 and L_2 with different implementations for a method m_2 along with their client in Figure 5. The former one always returns 2 while the latter returns the value of $x++$. It is easy to see that for any history from $\mathsf{history}([\![L_1]\!])$ there is an equivalent one from $\mathsf{history}([\![L_2]\!])$, so by definition $L_1 \sqsubseteq L_2$. However, Abstraction Theorem does not hold of L_1 and L_2. The subtlety here is that a client is able to perform a synchronisation that influences library's execution and makes it possible to detect a different behaviour of a linearized library.

In terms of our axiomatic model this means, that an execution of $C(L_2)$ violates RFMR' validity axiom, while $C(L_1)$ does not. To avoid this, each validity axiom that can be violated because of client's synchronisation contributes edges into a *deny* relation. This way any client synchronisation that breaks a library-local execution is explicitly forbidden.

5 Related work and conclusions

Recent work has proposed definitions of linearizability for the operational model of TSO [2] and the axiomatic model of C++11 [1]; the latter memory model is significantly more complex than TSO. The techniques we used in this paper are inspired by the construction of the definition of linearizability for C++11. By demonstrating their application in a simple and clean setting, we hope to highlight their main underlying ideas and make it easier for other researchers to use them for developing compositional reasoning methods for other memory models.

We also hope that the definition of linearizability for an axiomatic version of TSO will lend itself easier to automatic verification and testing than the operational definition [2]. Namely, model checking the latter requires enumerating an exponential number of concurrent program executions. In contrast, a single execution in an axiomatic model concisely represents whole classes of executions in a way that can be accepted by standard SAT or SMT solvers, which enables efficient verification [4].

References

[1] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. POPL, 2013. To appear.

[2] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, 2012.

[3] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 1990.

[4] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, 2012.

[5] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[6] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.