Reducing TSO to SC by Reduction

TSO Simplified

Ismail Kuru Serdar Tasiran
Koc University
ikuru@ku.edu.tr/stasiran@ku.edu.tr

Ali Sezgin
University of Cambridge
ali.sezgin@cl.cam.ac.uk

Abstract

A prominent way of analyzing programs written for relaxed memory models is to check whether it is sound, for the particular program under analysis, to assume sequential consistency (SC) which is perceived as the bare minimum for concurrent reasoning. The approach is based on establishing a *data race freedom* result, which essentially identifies for a given memory model the class of programs which cannot manifest non-SC behaviors. The total store ordering (TSO) memory model in particular has been fully characterized: a program will have a non-SC behavior if and only if it has a triangular data race. However, one is left to reason in the TSO world if the program fails to avoid those races.

In this paper, we develop a methodology which transforms a program with races to one without. Our framework is derived from QED, an abstraction-refinement approach based on Lipton's reduction theory. The main observation is that if one can systematically prove that each write to buffer and its associated flush from the buffer can be put together, then one can treat the whole program as SC. Putting together amounts to showing that actions have the correct mover types; e.g. each flush of a method is a left-mover.

The presentation of the framework is done in two phases. In the first phase, we formulate sufficient conditions under which a program can be transformed into another by using a class of abstraction operators. This formulation is done via an argument based on the whole set of executions of the program . In the second phase, how most of these ideas can be captured in a local reasoning style, very similar to QED. We illustrate our approach on various examples which are also mechanically verified by our tool, QED4TSO.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms program verification

Keywords reduction, total-store ordering, sequential consistency, race freedom, safety checking

1. Introduction

Both TSO and SC give the allusion that the thread-local program order is respected for memory accesses. Unlike SC where updates are assumed to take effect instantaneously over all threads, in TSO updates are observably split into two: a locally visible update and an instantaneous remote update not necessarily simultaneous with the local update. In formal models, this split is captured by a thread local queue in which local updates are inserted as soon as the local update occurs; entries are removed from the queue asynchronously and non-deterministically updating a single global shared memory. This queue is traditionally known as the store buffer.

It has been showed that it is possible to ignore the relaxed memory effects (relative to SC) if one adheres to a particular style of programming. Generally referred to under the heading data race freedom, these programs cannot distinguish whether they are running on an SC architecture or something weaker. If a relaxed memory model is proved to have this property, then proving correctness for data race free programs reduces to proving correctness under SC. For TSO, such a theorem was proven for a class of programs, called TRF (triangular race free): a program cannot distinguish SC from TSO iff it is TRF [**?**].

As for the programs that fall out of this class, and there are many more of them, the proof efforts are centered around the simplification of reasoning associated with the effect of the store buffer on the execution. Typically, one converts a TSO program into an equivalent one which has an additional unbounded buffer per thread modeling the asynchronous remote updates. Methodologies consequently differ on how they simplify the reasoning. It could involve reducing the points where asynchronous updates can happen [**?**] or bounding the size of the buffer [**?**]. So far no work has tried to cross the gap between these two classes of programs in a uniform manner.

2. Overview

In this section we are going to walk through several examples illustrating the main concepts of our approach. The discussion will be necessarily kept at a semi-formal level; all relevant formal definitions are given in the following sections. We start by explaining how one reasons about TSO programs using reduction.

Reduction for TSO. Consider a sequence of memory accesses done by two threads, t and u, given as:

$$\mathsf{W}_t(y/1) \cdot \mathsf{R}_t(x/0) \cdot \mathsf{W}_u(x/2)$$

This represents a sequence of a write to y by thread t, followed by a read to x again by thread, followed by a write to x by thread u.

To see why this is a *bad* (distinguishing) sequence for TSO, we will refer to an alternative characterization of memory accesses in TSO. We represent read accesses as usual, but change the way write accesses are represented. We split each write access W into two

```
Recv()              Send(d)
 while (R);          while (!R);
 d := Data;          Data := d;
 R := true;           R := false;
 return d;
```

Figure 1. A binary synchronous rendez-vous with two threads.

action (a,b)	$::=$	$\mathsf{W}^{\mathsf{l}}_t(x/v)$	(local write: insertion of the write of value v into location x by thread t)
	$\mid$	$\mathsf{W}^{\mathsf{r}}_t(x/v)$	(remote write: shared memory update of x with v by t)
	$\mid$	$\mathsf{R}_t(x/v)$	(a read of value v from x by t)
	$\mid$	B_t	(memory barrier by t)

Figure 2. The alphabet Σ of memory actions in TSO.

distinct accesses, W^{l} and W^{r}. Each sequence over simple memory accesses as given above will be identified with a set of sequences over the new alphabet such that i) each W is replaced with W^{l}, ii) each W^{l} has a matching W^{r} such that the thread local order among W^{l}'s is preserved by their associated W^{r}'s. For instance, the following sequence is one possible TSO-expansion of the SC-sequence above.

$$\mathsf{W}^{\mathsf{l}}_t(y/1) \cdot \mathsf{R}_t(x/0) \cdot \mathsf{W}^{\mathsf{l}}_u(x/2) \cdot \mathsf{W}^{\mathsf{r}}_u(x/2) \cdot \mathsf{W}^{\mathsf{r}}_t(y/1)$$

Observe that the ordering between the remote updates to y and x is the opposite of the ordering between their corresponding local updates. Intuitively this means that the order of updates to x and y as seen by t (y precedes x) and the order seen by a different thread v (x precedes y) will be different, which is impossible under SC.

We show that a similar characterization can be obtained using reduction arguments. An execution can be generated under SC if it is possible to *move* all local and their matching remote updates next to each other without changing the end state of the execution. For the example above, the question is whether it is possible to move either $\mathsf{W}^{\mathsf{r}}_t(y/\star)$ to the left of every possible concurrent action or $\mathsf{W}^{\mathsf{l}}_t(y/\star)$ to the right of every possible concurrent action. The trace above itself is not problematic because both remote writes are left-movers. In other words, the above trace is equivalent to the following:

$$\mathsf{W}^{\mathsf{l}}_t(y/1) \cdot \mathsf{W}^{\mathsf{r}}_t(y/1) \cdot \mathsf{R}_t(x/0) \cdot \mathsf{W}^{\mathsf{l}}_u(x/2) \cdot \mathsf{W}^{\mathsf{r}}_u(x/2)$$

However, in the presence of other writes such a transformation can become impossible. Consider for instance

$$\mathsf{W}^{\mathsf{l}}_t(y/1) \cdot \mathsf{R}_t(y/1) \cdot \mathsf{R}_t(x/0) \cdot \mathsf{W}^{\mathsf{l}}_u(x/2) \cdot \mathsf{R}_u(x/2) \cdot \mathsf{R}_u(y/0) \cdot \mathsf{W}^{\mathsf{r}}_u(x/2) \cdot \mathsf{W}^{\mathsf{r}}_t(y/1)$$

It is possible to move the local and remote writes to x done by thread u without changing the values read. However, it is impossible to make the local and remote writes to y done by thread t without changing one of the reads: if $\mathsf{W}^{\mathsf{r}}_t(y/1)$ is to the left of $\mathsf{W}^{\mathsf{l}}_u(x/2)$, the read of y by thread u will return 1 instead of 0; if the $\mathsf{W}^{\mathsf{l}}_t(y/1)$ is to the right of $\mathsf{W}^{\mathsf{l}}_u(x/2)$, the read of x by thread t will return 2 instead of 0.

Consider a simple template for a binary synchronous rendez-vous message passing as given in Fig. 1. There are two methods, `Recv` and `Send`. The receiver method `Recv` spins for the flag `R` until it is set to false. Once that happens, it reads the value stored in `Data`, resets `R` to true and returns the value it has read. The sender method `Send` operates dually: it spins on `R` until it is set to true, updates the value in `Data` and resets `R` to false.

Recall that a write can be treated as atomic if for any execution in which both its local and remote updates occur, one can find an equivalent execution in which they occur consecutively. For instance, if we want to show that the write to `R` done by `Recv` is atomic, we have to show that if the following is an execution

$$A \cdot \mathsf{W}^{\mathsf{l}}_t(R/\mathit{true}) \cdot B \cdot \mathsf{W}^{\mathsf{r}}_t(R/\mathit{true})$$

there is a partitioning of B into B_1 and B_2 such that

$$A \cdot B_1 \cdot \mathsf{W}^{\mathsf{l}}_t(R/\mathit{true}) \cdot \mathsf{W}^{\mathsf{r}}_t(R/\mathit{true}) \cdot B_2$$

is also an execution. One can construct such an execution because i) one can always move a local update to the right of other thread's actions, ii) there cannot be any remote updates to R in the sequence of accesses represented by B, iii) thus it is safe to move any reads of R done by a subsequent call to `Send` all of which will return true. Thus setting $B_1 = B$ and $B_2 = \varepsilon$ gives the desired execution. Similar arguments are used to prove that all the write accesses in Fig. 1 are atomic.

Abstracting memory accesses.

Transforming TSO into SC.

Local analysis for reduction. Once the program is transformed, one can appeal to the reduction theorems for TSO.

Our contributions are:

- Formal reasoning framework for TSO programs based on reduction.
- Systematic abstraction of a program so that the transformed has no SC-distinguishing executions and is free of assertion violations only if the original program is free of assertion violations.
- Structural transformation of a TSO program which has the same behavior under SC.
- Local and modular reduction framework for transformed programs.

3. Formal Framework

Notation. The symbols we are going to use and their meanings are given in Fig. 2. For simplicity, we omit locked operations which can be modeled by the given actions. A local write $\mathsf{W}^{\mathsf{l}}_t(x/v)$ *matches* the remote write $\mathsf{W}^{\mathsf{r}}_u(y/w)$ iff $t = u$, $x = y$ and $v = w$. We will use the notation $\Sigma_{op,thr,loc}$ to denote the subset of Σ which contains all actions of type $op \in \{\mathsf{W}^{\mathsf{l}}, \mathsf{W}^{\mathsf{r}}, \mathsf{R}, \mathsf{B}\}$, by thread thr, and into location loc. Omitting parameters denotes existential quantification; e.g. $\Sigma_{-,t,-}$ is the set of all actions done by thread t.

Let $\mathbf{e}$ be a sequence over Σ. We will use indexed notation to refer to the elements in $\mathbf{e}$: $\mathbf{e}[i]$ is the i^{th} action in $\mathbf{e}$. Similarly, we let $\mathbf{e}\langle i,j\rangle$ denote the segment of $\mathbf{e}$ from $\mathbf{e}[i]$ to $\mathbf{e}[j]$ with both ends inclusive. The length of $\mathbf{e}$ is written as $|\mathbf{e}|$ and gives the number of actions in $\mathbf{e}$. Let π be a permutation over $[1,k]$ and $\mathbf{e}$ be of length k. We use $\pi(\mathbf{e})$ to denote the sequence $\mathbf{e}[\pi(1)] \dots \mathbf{e}[\pi(k)]$. Two executions $\mathbf{e}$ and $\mathbf{f}$ are *permutationally equivalent*, written $\mathbf{e} \sim_\pi \mathbf{f}$, if there is a permutation π such that $\mathbf{e} = \pi(\mathbf{f})$.

The projection of $\mathbf{e}$ into $\Sigma_{o,t,l}$, written as $\mathbf{e} \downarrow_{o,t,l}$, is the subsequence obtained by keeping only those actions in $\Sigma_{o,t,l}$. For instance, $\mathbf{e} \downarrow_{\mathsf{W}^{\mathsf{l}},t,-}$ is the subsequence of $\mathbf{e}$ consisting of all the local writes done by t.

TSO-executions. Let $\mathbf{e}$ be a sequence over Σ. It is *matched* if for any t there is an injection μ from $\mathbf{e} \downarrow_{\mathsf{W}^{\mathsf{r}},t,-}$ to $\mathbf{e} \downarrow_{\mathsf{W}^{\mathsf{l}},t,-}$ such that $\mu(a) = b$ implies that b matches a. A local write $\mathbf{e}[i]$ is *buffered at* j if it is matched by a remote write at position $l > j$. We will call a matched $\mathbf{e}$ *complete* if there are no buffered writes at $|\mathbf{e}|$.

A TSO-execution is a matched sequence $\mathbf{e}$ over Σ subject to well-formedness constraints:

- The order of remote writes per thread respects the order of their matching local writes. Formally, for $i < j$ such that $\mathbf{e}[i] = \mathsf{W}^{\mathsf{l}}_t(x/v)$ and $\mathbf{e}[j] = \mathsf{W}^{\mathsf{l}}_t(y/w)$ and there is k with $\mu(\mathbf{e}[k]) = \mathbf{e}[j]$, then there is l such that $j < l < k$ and $\mu(\mathbf{e}[l]) = \mathbf{e}[i]$.
- The read values are consistent. Formally, if $\mathbf{e}[j] = \mathsf{R}_t(x/v)$, then either i) $\mathsf{W}^{\mathsf{l}}_t(x/v)$ is the most recent buffered write at j by t to x, or ii) no buffered write to x by t at j exists and $\mathsf{W}^{\mathsf{r}}_u(x/v)$ is the most recent remote write, or iii) neither condition applies and v is the initial value.
- The barrier operations can only happen when the buffer is empty. Formally, if $\mathbf{e}[j] = \mathsf{B}_t$ and if $\mathsf{W}^{\mathsf{l}}_u(x/v)$ is buffered at j, then $t \neq u$.

DEFINITION 1 (Atomic). *A local write $\mathbf{e}[i]$ is* atomic *if it is not buffered at $i+1$. A complete TSO-execution $\mathbf{e}$ is called* atomic *if all of its local writes are atomic.*

Each TSO-execution $\mathbf{e}$ induces a partial order $<^{tso}_{\mathbf{e}}$ over $\{\mathbf{e}[i] \mid i \in [1, |\mathbf{e}|]\}$ such that $a <^{tso}_{\mathbf{e}} b$ if a occurs before b in $\mathbf{e}$, $a, b \in \Sigma_{-,t,-}$ and one of the following holds:

- $a \in \Sigma_{\mathsf{W}^{\mathsf{r}},-,-}$ iff $b \in \Sigma_{\mathsf{W}^{\mathsf{r}},-,-}$.
- $b = \mu(a)$.

SC-executions. An SC-execution $\mathbf{s}$ is a sequence over the alphabet $\{\mathsf{W}_t(x/v)\} \cup \{\mathsf{R}_t(x/v)\}$. The actions of the form $\mathsf{W}_t(x/v)$, denoting the write of value v into location x by thread t, are called *write* actions. Each SC-execution has to satisfy: if $\mathbf{s}[i] = \mathsf{R}_t(x/v)$, then either i) there is $j < i$ with $\mathbf{s}[j] = \mathsf{W}_u(x/v)$ and there are no write actions to x in $\mathbf{s}\langle j+1, i\rangle$, or ii) there is no write action in $\mathbf{s}\langle 1, i\rangle$ and v is $\perp$.

Similar to TSO-executions, each SC-execution $\mathbf{s}$ induces a partial order $<^{sc}_{\mathbf{s}}$ such that $a <^{sc}_{\mathbf{s}} b$ if a occurs before b in $\mathbf{s}$ and both belong to the same thread.

Correspondence between TSO and SC. We define a mapping $\ulcorner\urcorner$ from the actions of TSO to those of SC as follows.

$$\ulcorner a \urcorner \overset{def}{=} \begin{cases} \mathsf{W}_t(x/v) & \text{, if } a = \mathsf{W}^{\mathsf{l}}_t(x/v) \\ \mathsf{R}_t(x/v) & \text{, if } a = \mathsf{R}_t(x/v) \\ \varepsilon & \text{, if } a = \mathsf{W}^{\mathsf{r}}_t(x/v) \\ \varepsilon & \text{, if } a = \mathsf{B}_t \end{cases}$$

For a TSO-execution $\mathbf{e}$, let $\ulcorner \mathbf{e} \urcorner$ denote the sequence obtained by the point-wise extension of $\ulcorner\urcorner$.

Let $\mathbf{e}$ be a complete TSO-execution. We define $\mathsf{SC}(\mathbf{e})$ to be the set of all SC-executions $\mathbf{s}$ such that $\ulcorner \mathbf{e} \downarrow_{-,t,-} \urcorner = \mathbf{s} \downarrow_{-,t,-}$. Informally, $\mathbf{s}$ belongs to $\mathsf{SC}(\mathbf{e})$ if it is an SC-execution and respects the thread local ordering of read and local write actions.

A particular subset of $\mathsf{SC}(\mathbf{e})$ is of interest to us. Let $\mathsf{SC}_S(\mathbf{e}) \subseteq \mathsf{SC}(\mathbf{e})$ be the set of all $\mathbf{s}$ such that $\mathbf{e} \downarrow_{\mathsf{W}^{\mathsf{r}},-,-} [i] = \mathsf{W}^{\mathsf{r}}_t(x/v)$ iff $\mathbf{s} \downarrow_{\mathsf{W},-,-} [i] = \mathsf{W}_t(x/v)$. Informally, $\mathbf{s}$ will be in $\mathsf{SC}_S(\mathbf{e})$ if additionally the order among the write actions in $\mathbf{s}$ preserves the order among the remote write actions in $\mathbf{e}$. We have the following result following immediately from definitions.

PROPOSITION 1. *For any atomic TSO-execution* $\mathbf{e}$, $\ulcorner \mathbf{e} \urcorner \in \mathsf{SC}_S(\mathbf{e})$.

Proposition 1 implies that an atomic TSO-execution can always be transormed into an SC-execution by simply replacing each (adjacent) pair of local and remote write actions by their image under $\ulcorner\urcorner$.

Equivalence ($\approx$) and partial order ($\sqsubseteq$) over TSO-executions. Two complete TSO-executions $\mathbf{e}$ and $\mathbf{f}$ are *equivalent*, written $\mathbf{e} \approx \mathbf{f}$, if one can be obtained from the other by moving the actions such that the per thread order is preserved. Formally, $\mathbf{e} \approx \mathbf{f}$ if $\mathbf{e} \sim_\pi \mathbf{f}$ and $\mathbf{e} \downarrow_{-,t,-} = \mathbf{f} \downarrow_{-,t,-}$. They are *strictly* equivalent, written $\mathbf{e} \approx_S \mathbf{f}$, if the order among the remote writes is also preserved. Formally, $\mathbf{e} \approx_S \mathbf{f}$ if $\mathbf{e} \approx \mathbf{f}$ and $\mathbf{e} \downarrow_{\mathsf{W}^{\mathsf{r}},-,-} = \mathbf{f} \downarrow_{\mathsf{W}^{\mathsf{r}},-,-}$. Whenever no confusion is likely to arise, we will use π to denote one of the permutations between two (strictly) equivalent TSO-executions establishing their (strict) equivalence.

Atomicity induces a partial order $\sqsubseteq$ on TSO-equivalent traces. A TSO-execution $\mathbf{e}$ is *tighter* than $\mathbf{f}$, written $\mathbf{e} \sqsubseteq \mathbf{f}$ if $\mathbf{e} \approx \mathbf{f}$ and whenever $\mathbf{f}[i]$ is an atomic local write, then so is $\mathbf{e}[\pi^{-1}(i)]$. In other words, $\mathbf{e}$ is tighter than an equivalent $\mathbf{f}$ if all the atomic writes of the latter are also atomic in the former. The execution $\mathbf{e}$ is strictly tighter than $\mathbf{f}$, written $\mathbf{e} \sqsubseteq_S \mathbf{f}$, if $\mathbf{e}$ is tighter than $\mathbf{f}$ and they are strictly equivalent.

Each TSO-execution thus induces a set of (strictly) tighter executions. Let $\mathsf{T}(\mathbf{e}) = \{\mathbf{f} \mid \mathbf{f} \sqsubseteq \mathbf{e}\}$; that is, the set of all TSO-executions tighter than $\mathbf{e}$. Let $\mathsf{T}_S(\mathbf{e})$ denote the subset of $\mathsf{T}(\mathbf{e})$ consisting of only strictly equivalent executions.

DEFINITION 2 (SC-like). *Let $\mathbf{e}$ be a TSO-execution. It is called* observationally SC-like *if $\mathsf{T}(\mathbf{e})$ contains an atomic execution. It is called* SC-like *if $\mathsf{T}_S(\mathbf{e})$ contains an atomic execution.*

Since for any TSO-execution $\mathbf{e}$ we have $\mathsf{T}_S(\mathbf{e}) \subseteq \mathsf{T}(\mathbf{e})$, SC-like is a stronger property than observationally SC-like. The terms are not arbitrarily named as the following proposition shows.

PROPOSITION 2. *Let $\mathbf{e}$ be a complete TSO-execution. It is observationally SC-like iff $\mathsf{T}(\mathbf{e}) \cap \mathsf{SC}(\mathbf{e}) \neq \emptyset$. It is SC-like iff $\mathsf{T}_S(\mathbf{e}) \cap \mathsf{SC}_S(\mathbf{e}) \neq \emptyset$.*

Conflict Relations. Two TSO actions a and b *conflict*, written $\mathsf{Conf}(a, b)$, if one of the following holds:

$(\mathbf{Rem})$ $a, b \in \Sigma_{\mathsf{W}^{\mathsf{r}},t,-}$.

$(\mathbf{Thr})$ $a, b \in \Sigma_{\{\mathsf{W}^{\mathsf{l}},\mathsf{R},\mathsf{B}\},t,-}$.

$(\mathbf{Loc})$ $a, b \in \Sigma_{-,-,x}$ and $\{a, b\} \not\subseteq \Sigma_{\mathsf{R},-,-}$.

They *strictly* conflict, $\mathsf{Conf}_S(a, b)$, if $(\mathbf{Rem})$ is replaced with

$(\mathbf{RemS})$ $a, b \in \Sigma_{\mathsf{W}^{\mathsf{r}},-,-}$.

Let $\mathbf{e}$ be a TSO-execution of length k. A sequence $\mathbf{f}$ is a *shuffling* of $\mathbf{e}$ if $\mathbf{e} \sim_\pi \mathbf{f}$ and there is some i such that for all $j \notin \{i, i+1\}$, $\mathbf{e}[j] = \mathbf{f}[j]$ and $\mathbf{e}[i] = \mathbf{f}[i+1]$. The position i is called the *pivot* (of shuffling). A shuffling of $\mathbf{e}$ with pivot i is *valid* if $\pi(\mathbf{e})$ is a TSO-execution and $\mathbf{e}[i]$ does not conflict with $\mathbf{e}[i+1]$. Let $\mathsf{Shuff}(\mathbf{e})$ denote the closure of the valid shufflings on $\mathbf{e}$. Formally, $\mathsf{Shuff}(\mathbf{e})$ is defined as the set satisfying

- $\mathbf{e} \in \mathsf{Shuff}(\mathbf{e})$,
- $\mathbf{f} \in \mathsf{Shuff}(\mathbf{e})$ implies there is $\mathbf{g} \in \mathsf{Shuff}(\mathbf{e})$ and $\mathbf{f}$ is a valid shuffling of $\mathbf{g}$.

Observe that there are two further constraints implicitly enforced by requiring a valid shuffling with pivot i. First, if $\mathbf{e}[i+1]$ is a barrier action by thread t and $\mathbf{e}[i]$ is a remote write action by the same thread, then they cannot be reordered as barrier cannot occur when there is a buffered write. Second, if $\mathbf{e}[i]$ is a local write action by t and $\mathbf{e}[i+1]$ is the matching remote write action, reordering them is not allowed since the resulting sequence will not be a TSO-execution.

The shuffling closure of $\mathbf{e}$ gives an operational under-approximation of TSO-execution equivalence $\approx$ as the following result indicates.

PROPOSITION 3. *For complete TSO-execution* $\mathbf{e}$, $\mathsf{Shuff}(\mathbf{e}) \subseteq [\mathbf{e}]_\approx$.

$$\begin{array}{lll}
P & ::= & M, P \mid \varepsilon \\
M\,(m,m',\ldots) & ::= & T\ N(T)\ \{C\} \\
C\,(c,c',\ldots) & ::= & S\,;C \mid \varepsilon \\
S\,(s,s',\ldots) & ::= & R := \mathtt{mem}[E] \mid \mathtt{mem}[E] := R \mid \\
 & & R := E \mid \mathtt{fence} \mid \\
 & & \mathtt{if}\ E\ \mathtt{then}\ \{C\}\ \mathtt{else}\ \{C\} \mid \\
 & & \mathtt{while}(E)\ \{C\} \mid \mathtt{atomic}\{C\} \\
 & & \mathtt{assume}\ E \mid \mathtt{assert}\ E \mid \mathtt{skip} \\
T & ::= & \langle Bool\rangle \mid \langle Int\rangle \\
N & ::= & \langle Name\rangle \\
R\,(r,r',\ldots) & ::= & \mathtt{r}[i]\ (i \in \mathbb{N}) \\
E\,(e,e',\ldots) & ::= & R \mid i \mid N(T) \mid \mathtt{tid} \mid E+E \mid E-E \mid \ldots \\
 & & E \wedge E \mid E \vee E \mid \neg E \mid \ldots
\end{array}$$

Figure 3. A simple programming language.

3.1 Programming Language

In this subsection, we formalize the programming language we are going to use in the rest of the paper.

Syntax. We use a simple programming language, whose grammar is given in Fig. 3. The nonterminal R, ranged over by r, refers to *registers* and are used to model thread local address space. The nonterminal E, ranged over by e, refers to well-formed *expressions* that can be written using registers, mathematical and logical operators, and the return values of procedures. The shared data space is mapped by $\mathtt{mem}[e]$, where e is an expression which evaluates to $\mathbb{N}$. A Instructions can be used to read from memory ($r := \mathtt{mem}[e]$), update the contents of a memory location ($\mathtt{mem}[e] := e'$), empty the store buffer (fence), assign a value to a register ($r := e$). The model control flow, we have the usual branching ($\mathtt{if}\ e\ \mathtt{then}\ \{e_1\}\ \mathtt{else}\ \{e_2\}$) and looping ($\mathtt{while}(e)\ \{\}$) instructions. Additionally, we will also use an explicit blocking instruction (assume e), which blocks as long as e evaluates to *false*. Finally, the instruction (assert e) is used to claim that e should hold whenever this instruction can execute. Let $\mathsf{Stmt}(P)$ denote the set of statements used in program P.

Operational Semantics. A *labelled program* is a pair (P, Lab) consisting of a program P and a *labelling function* $\mathsf{Lab} : \mathsf{Stmt}(P) \mapsto L$ mapping each statement of P to a unique *label* in the set of labels, L. When no confusion is likely to arise, we use P to refer also to a labelled program in which case Lab is implicit. We use a labelled transition system (LTS) to define the semantics of a labelled program under either TSO or SC.

Two statements s, s' in P are *consecutive* if $s; s'$ appears in P. In such a case, s' is the *sequential successor* of s and s the *sequential predecessor* of s', denoted as $s' = \mathsf{Succ}(s)$ and $s = \mathsf{Prec}(s')$. If s has no sequential successor, we let $\mathsf{Succ}(s) = \varepsilon$. A *code block* is any code (C) delimited by two matching braces. The unique first statement in a non-empty code block is referred to as $\mathsf{Fst}(C)$. Observe that if s is the last statement in a code block, then $\mathsf{Succ}(s) = \varepsilon$.

A *program state* of P is a tuple $(\mathsf{Ctrl}, \mathsf{Val}, \mathsf{Buf}, \mathsf{lck})$. The *control flow* $\mathsf{Ctrl} : T \mapsto L^*$ maps each thread in the set of thread identifiers T to some sequence over labels. Intuitively, Ctrl encodes the execution context for each thread in the program. If $\mathsf{Ctrl}(t) = \vec{l}$, then $\vec{l}\langle 1, |\vec{l}| - 1\rangle$ keeps the nesting history and $\vec{l}[|\vec{l}|]$ is the label of the next statement to be executed by thread t. The *valuation* $\mathsf{Val} : (\mathbb{N} \cup T \times R) \mapsto \mathbb{N}$ maps each memory location and register of each thread to its content, assumed to be a non-negative integer. Intuitively, Val holds the storage state: the contents of each register of each thread and the contents of the (single) global memory. The contents of a memory location i, $\mathtt{mem}[i]$, is accessed by $\mathsf{Val}(i)$, and the content of some register r used by a thread t is accessed by $\mathsf{Val}(t, r)$. The *buffer view* $\mathsf{Buf} : T \mapsto (\mathbb{N} \times \mathbb{N})^*$ maps each thread to a sequence over pairs of natural numbers. Intuitively, $\mathsf{Buf}(t) = (a_1, d_1) \ldots (a_k, d_k)$ means that the store buffer of thread t contains k buffered writes, the oldest being to location a_1 of value d_1 and the newest being to location a_k of value d_k. We use $a \in \mathsf{Buf}(t)$ to denote the fact that there is $j \in [1, k]$ with $a_j = a$. In such a case, let $\mathsf{RdBuf}(t, a)$ denote d_j such that $a_j = a$ and for all $i > j$, $a_i \neq a$. Finally, the *lock state* lck is a variable ranging over $\{-1, -2\} \cup T$. Intuitively, lck is -1 if no thread is currently executing an atomic block, is t if thread t is executing an atomic block. When lck is set to -2, it denotes an assertion violation on occurrence of which all threads are blocked. The transition relation of program P is given in Fig. 4.

Program runs. A *run* of program P is an alternating sequence of states and transition labels conforming to the operational semantics of Fig. 4. Formally, a run $\vec{r}$ is a sequence $q_0 t_1 q_1 \ldots t_k q_k$, where q_i's are states of $LTS(P)$ and for each $i \in [1, k]$, $q_{i-1} \xrightarrow{t_i} q_i$ is a transition of $LTS(P)$. The sequence of transitions $t_1 \ldots t_k$ is called the *trace* of the run, denoted by $\mathsf{Tr}(\vec{r})$.

A program run is *TSO compliant* if it does not contain the WR transition. Similarly, a program run is *SC compliant* if it does not the WRB, WRM transitions. For notational convenience we leave FNC rules in SC compliant runs. It is easy to show that in SC compliant runs fence and skip are interchangeable. Let $\mathbf{R}_{tso}(P)$ denote the set of all TSO compliant runs of P. Similarly, let $\mathbf{R}_{sc}(P)$ denote the set of all SC compliant runs of P.

Memory traces. In order to relate program runs to executions of the previous section, we define a mapping $\mathsf{Mem} : Q^P \times Tr^P \to \Sigma$ in Fig. 5.

Let $\vec{r} = q_0 t_1 q_1 \ldots t_k q_k$ be a run. The memory trace of $\vec{r}$ is the sequence $\mathsf{Mem}(q_0, t_1) \cdot \mathsf{Mem}(q_1, t_2) \cdot \ldots \cdot \mathsf{Mem}(q_{k-1}, t_k)$. With an abuse in notation, we let $\mathsf{Mem}(\vec{r})$ denote the memory trace of $\vec{r}$. The following result establishes the link between runs and executions.

PROPOSITION 4. *Let $\vec{r}$ be a TSO (resp. SC) compliant run. Then $\mathsf{Mem}(\vec{r})$ is a TSO-execution (resp. SC-execution).*

4. Reduction for TSO

In this section, we explain how the reduction theory of Lipton can be used for TSO. Our goal is to present sufficient conditions for programs such that when a program satisfies these conditions the program is guaranteed to be unable to distinguish TSO semantics from SC semantics. This is the first step of applying reduction to TSO programs. In the following sections, we will show how we can extend the applicability of the same approach for programs which are initially TSO distinguishing.

DEFINITION 3 (Movers). *Let P be a labelled program and let $\simeq$ be an equivalence relation over $\mathbf{R}_{tso}(P)$. Let s be a statement that occurs in some method's body m of P. Then, s is* left-mover in $\simeq$ *if for any $\vec{r} = \vec{r}[1] \ldots \vec{r}[k] \in \mathbf{R}_{tso}(P)$ we have $\vec{r}[i]\vec{r}[i+1] = (R', t' : s')(R, t : s)$ such that $\mathsf{Mem}(s') <^{tso}_{\mathsf{Mem}(\vec{r})} \mathsf{Mem}(s)$ does not hold, then there exists a run $\vec{r'}$ in $[\vec{r}]_{\simeq}$ such that $\vec{r'}\langle 1, i\rangle = \vec{r}\langle 1, i-1\rangle\vec{r}[i+1]$.*

Similarly, s is right-mover in $\simeq$ *if for any $\vec{r} = \vec{r}[1] \ldots \vec{r}[k] \in \mathbf{R}_{tso}(P)$ we have $\vec{r}[i-1]\vec{r}[i] = (R, t : s)(R', t' : s')$ such that $\mathsf{Mem}(s) <^{tso}_{\mathsf{Mem}(\vec{r})} \mathsf{Mem}(s')$ does not hold, then there exists a run $\vec{r'}$ in $[\vec{r}]_{\simeq}$ such that $\vec{r'}\langle 1, i\rangle = \vec{r}\langle 1, i-2\rangle\vec{r}[i]\vec{r}[i-1]$.*

Intuitively, s is left mover (resp. right mover) if reordering s before (resp. after) any other statement that is concurrent with s does not change the behavior (all runs belonging to the same equivalence have the same behavior). In the classic definition of

$$\boxed{(\mathsf{Ctrl}, \mathsf{Val}, \mathsf{Buf}, \mathsf{lck}) \xrightarrow{Rl,t:s} (\mathsf{Ctrl}', \mathsf{Val}', \mathsf{Buf}', \mathsf{lck}')}$$

$$\frac{Rl = \textsc{Init} \qquad M_i \in P \qquad M_i = n(a)\ \{C\} \qquad \mathsf{Ctrl}(t) = \varepsilon}{\mathsf{Val}[(t, \mathtt{r}_{in}) \mapsto a] \qquad \mathsf{Ctrl}[t \mapsto \mathsf{Fst}(C)]}$$

$$\frac{\begin{array}{c} Rl = \textsc{Rd} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = r := \mathtt{mem}[e] \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Val}[(t, r) \mapsto \mathsf{Val}(\mathsf{Val}\llbracket e \rrbracket t)] \qquad \mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{\begin{array}{c} Rl = \textsc{RdM} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = r := \mathtt{mem}[e] \qquad \mathsf{Val}\llbracket e \rrbracket t \notin \mathsf{Buf}(t) \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Val}[(t, r) \mapsto \mathsf{Val}(\mathsf{Val}\llbracket e \rrbracket t)] \qquad \mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{\begin{array}{c} Rl = \textsc{RdB} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = r := \mathtt{mem}[e] \qquad \mathsf{Val}\llbracket e \rrbracket t \in \mathsf{Buf}(t) \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Val}[(t, r) \mapsto \mathsf{RdBuf}(t, \mathsf{Val}\llbracket e \rrbracket t)] \qquad \mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{\begin{array}{c} Rl = \textsc{Wr} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{mem}[e] := r \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Val}[(t, r) \mapsto \mathsf{Val}\llbracket e \rrbracket t] \qquad \mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{\begin{array}{c} Rl = \textsc{WrB} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{mem}[e] := r \qquad \mathsf{Buf}(t) = \vec{b} \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Buf}[t \mapsto \vec{b} \cdot (\mathsf{Val}\llbracket e \rrbracket t, \mathsf{Val}(t, r))] \qquad \mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{Rl = \textsc{WrM} \qquad \mathsf{Buf}(t) = (a, d) \cdot \vec{b} \qquad \mathsf{Enb}(t)}{\mathsf{Buf}[t \mapsto \vec{b}] \qquad \mathsf{Val}[a \mapsto d]}$$

$$\frac{\begin{array}{c} Rl = \textsc{WrR} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = r := e \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Val}[(t, r) \mapsto \mathsf{Val}\llbracket e \rrbracket t] \qquad \mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{\begin{array}{c} Rl = \textsc{Fnc} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{fence} \qquad \mathsf{Buf}(t) = \varepsilon \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{\begin{array}{c} Rl = \textsc{IfT} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{if}\ e\ \mathtt{then}\ \{c_1\}\ \mathtt{else}\ \{c_2\} \qquad \mathsf{Val}\llbracket e \rrbracket t = \mathit{true} \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l') \cdot \mathsf{Fst}(c_1)]}$$

$$\frac{\begin{array}{c} Rl = \textsc{IfE} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{if}\ e\ \mathtt{then}\ \{c_1\}\ \mathtt{else}\ \{c_2\} \qquad \mathsf{Val}\llbracket e \rrbracket t = \mathit{false} \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l') \cdot \mathsf{Fst}(c_2)]}$$

$$\frac{\begin{array}{c} Rl = \textsc{WhI} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{while}(e)\ \{c\} \qquad \mathsf{Val}\llbracket e \rrbracket t = \mathit{true} \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Ctrl}[t \mapsto \vec{l} \cdot l' \cdot \mathsf{Fst}(c)]}$$

$$\frac{\begin{array}{c} Rl = \textsc{WhE} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{while}(e)\ \{c\} \qquad \mathsf{Val}\llbracket e \rrbracket t = \mathit{false} \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{Rl = \textsc{Skp} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \qquad s = \mathtt{skip}}{\mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{\begin{array}{c} Rl = \textsc{AtB} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{atomic}\{c\} \qquad \mathsf{lck} = -1 \end{array}}{\mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l') \cdot l_x \cdot \mathsf{Fst}(c)] \qquad \mathsf{lck} = t}$$

$$\frac{Rl = \textsc{AtE} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l_x \qquad \mathsf{lck} = t}{\mathsf{Ctrl}[t \mapsto \vec{l}] \qquad \mathsf{lck} = -1}$$

$$\frac{\begin{array}{c} Rl = \textsc{AsT} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{assert}\ e \qquad \mathsf{Val}\llbracket e \rrbracket t = \mathit{true} \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

$$\frac{\begin{array}{c} Rl = \textsc{AsF} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{assert}\ e \qquad \mathsf{Val}\llbracket e \rrbracket t = \mathit{false} \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{lck} = -2}$$

$$\frac{\begin{array}{c} Rl = \textsc{Asm} \qquad \mathsf{Ctrl}(t) = \vec{l} \cdot l' \qquad \mathsf{Lab}(s) = l' \\ s = \mathtt{assume}\ e \qquad \mathsf{Val}\llbracket e \rrbracket t = \mathit{true} \qquad \mathsf{Enb}(t) \end{array}}{\mathsf{Ctrl}[t \mapsto \vec{l} \cdot \mathsf{Succ}(l')]}$$

Figure 4. Operational semantics for TSO and SC.

reduction, the equivalence relation requires that the two sequences be permutations of each other and that the end states are identical, which corresponds to $\approx_S$ in the TSO context. The reason for the added generality will become clear once we generalize reduction with abstraction.

Our first result about movers follows immediately from definitions.

LEMMA 1. *Let P be a labelled program. All of its TSO runs are SC-like if all remote writes are left-movers in $\approx_S$. Dually, all of its TSO runs are SC-like if all local actions (excluding remote writes) are right-movers in $\approx_S$.*

All of its TSO runs are observationally SC-like if all remote writes are left-movers in $\approx$. Dually, all of its TSO runs are observationally SC-like if all local actions (excluding remote writes) are right-movers in $\approx$.

This result depends on a strong constraint which is unlikely to be satisfied by many programs. In what follows we will provide a series of incrementally more general results. Fix a labelled program $P = \{m_1, \ldots, m_n\}$.

$$\mathsf{Mem}(q,(Rl,t:s)) \overset{def}{=} \begin{cases} \mathsf{R}_t(q.\mathsf{Val}[\![e]\!]t/q.\mathsf{Val}(q.\mathsf{Val}[\![e]\!]t)) & ,\ Rl \in \{\textsc{Rd},\textsc{RdM}\},\ s = r := \mathtt{mem}[e] \\ \mathsf{R}_t(q.\mathsf{Val}[\![e]\!]t/q.\mathsf{RdBuf}(t,q.\mathsf{Val}[\![e]\!]t)) & ,\ Rl = \textsc{RdB},\ s = r := \mathtt{mem}[e] \\ \mathsf{W}_t(q.\mathsf{Val}[\![e]\!]t/q.\mathsf{Val}(t,r)) & ,\ Rl = \textsc{Wr},\ s = \mathtt{mem}[e] := r \\ \mathsf{W}^{\mathsf{l}}_t(q.\mathsf{Val}[\![e]\!]t/q.\mathsf{Val}(t,r)) & ,\ Rl = \textsc{WrB} \\ \mathsf{W}^{\mathsf{r}}_t(a/d) & ,\ Rl = \textsc{WrM},\ q.\mathsf{Buf}(t) = (a,d)\cdot\vec{b} \\ \mathsf{B}_t & ,\ Rl = \textsc{Fnc} \\ \varepsilon & ,\ \text{otherwise} \end{cases}$$

Figure 5. Mapping transitions to memory operations.

LEMMA 2. *All TSO runs of P are SC-like if for each method m_i, either all of its local actions are right-movers in $\approx_S$ or all of its remote write actions are left-movers.*

This result helps us to reason about methods individually. We have the following corollary, which is also implied by triangular-race freedom of [**?**].

COROLLARY 1. *Let P be such that each m_i either only updates the shared memory (no read actions) or only reads shared memory (no write actions). Then all of its TSO runs are SC-like.*

Now we will investigate the impact of placing fence statements in restricting the non-SC-like behaviors of programs. For the following fix s^{loc} and s as matching local and remote write actions.

LEMMA 3. *If in any TSO run in which s^{loc} is executed by some thread t, all the combined write actions executed by t up to the occurrence of s^{loc} are atomic and s is left-mover, then s^{loc} and s are atomic.*

This naturally leads to the following special instance which gives an insight about how fence statements lead to SC-like programs.

COROLLARY 2. *If in any TSO run in which s^{loc} is executed by thread t, there exists a fence action executed by t preceding s^{loc}, no other local write actions by t occur between the two actions and s is left-mover, then s^{loc} and s are atomic.*

This means that, unlike the general case, in order to argue that a write immediately following a fence statement is atomic, we only need to prove that its matching remote write action is left-mover. There is a dual of this result which we state next.

LEMMA 4. *If in any TSO run in which s^{loc} is executed by thread t, there does not exist any read action executed by t until the occurrence of s, then s^{loc} is right-mover and s^{loc} and s are atomic.*

The next result which is a special case of the previous result explains why the use of fence statements restricts the runs of a TSO program to SC-like behaviors.

COROLLARY 3. *Let $s; C;$ `fence` be a code block in some m_i such that C does not contain any read of shared memory (no statements of the form $r := \mathtt{mem}[e]$). Then s and all the writes in C are atomic.*

CODE SNIPPETS DEMONSTRATING THE RESULTS.

5. Abstracting TSO programs

The main problem with the existing body of work on TSO program verification is the impossibility of handling programs which do contain non SC-like runs, i.e. programs with triangular races. Our approach so far allows us to at least alleviate some of the difficulties in reasoning by showing that certain write statements can be taken to be atomic. In this section, we go one step further and show abstraction can be used to turn a non-SC-like TSO program into one that is SC-like.

INSERTASSERT	$s \rightsquigarrow \mathtt{atomic}\{\mathtt{assert}\ e; s\}$
WEAKASSUME	$\mathtt{assume}\ e \rightsquigarrow \mathtt{assume}\ e'$ {provided $e \Rightarrow e'$}
VALNONDET	$r := \mathtt{mem}[e] \rightsquigarrow r := \star$ $r := e \rightsquigarrow r := \star$ $\mathtt{mem}[e] := r \rightsquigarrow \mathtt{mem}[e] := \star$
CTRLNONDET	$s \rightsquigarrow$ $\mathtt{if}\ \star$ $\mathtt{then}\ \{\mathtt{assume}\ e; s'\}$ $\mathtt{else}\ \{\mathtt{assume}\ e'; s''\}$ {provided $e \vee e'$ is tautology, and s' and s'' are abstractions of s}

Table 1. Substitution rules guaranteeing sound abstraction.

The use of abstraction in reduction was successfully demonstrated in [**?**] in the context of sequentially consistent programs. Since the soundness of the method crucially depends on the atomicity of each action, it was not clear how one can adopt those techniques to weaker memory models. Here we address and resolve the issue of non-atomic writes for TSO.

DEFINITION 4 (Program Abstraction). *Let P and P' be two programs. We say that P'* abstracts *P, if one of the following holds:*

- *P' has a failed run, or*
- *P does not have a failed run and for each terminated run $\vec{r} \in \mathbf{R}_{tso}(P)$, there exists a run $\vec{r'} \in \mathbf{R}_{tso}(P')$ such that $\mathsf{Mem}(\vec{r}) = \mathsf{Mem}(\vec{r'})$.*

Intuitively, P' abstract P if P' contains an assertion violation or has more behaviors than P. This means that if P' can be proven to contain no assertion violations, then neither does P. We should note that the other direction, that when P does not contain an assertion violation neither should P', does not hold in general.

Abstraction rules. There are many ways to ensure that a syntactic manipulation of P results in another program P' abstracting the former. In Table 1, we list several rules which are essentially individual statement replacements that provide a sound abstraction.

The first substitution rule, INSERTASSERT, is the insertion of an assertion to any statement. Recall that an assert statement either does nothing (if its predicate evaluates to true) or results in a failed run. Both cases trivially satisfy the conditions of abstraction.

A dual rule is to weaken assumptions, the rule WEAKASSUME. In this case, the assume statement with a weaker predicate will allow for more behaviors.

Another rule, VALNONDET, introduces non-deterministic reads or writes. Let $\star$ be an expression that can evaluate to any integer value. [1] Then replacing any expression with e with $\star$ is another abstraction.

[1] Essentially, with the introduction of $\star$ we are changing the evaluation operator $[\![]\!]$ from a mapping from expressions to values to a mapping from expression to sets of values.

Finally, the rule CtrlNondet, introduces non-deterministic control flow. The idea is to replace a statement s by an if-then-else statement such that the branches are not necessarily mutually exclusive, i.e. certain states may satisfy both e and e' of the rule in Table 1. This is a sound abstraction because no matter which branch is taken (and at any state at least one of these branches are enabled) whatever s was doing, possibly more, in the original program will be done.

For a detailed exposition of how abstraction in conjunction with reduction leads to a natural style of reasoning in safety proofs, the reader is referred to [**?**]. In the remainder of this section, we work through an example program which contains a triangular race and hence is not initially amenable to an SC analysis. We show how abstracting the program leads to a program which is SC-like.

```
DoubleCheckInit()
{
 r:=mem[Obj];
 if r=0 then
 {
  atomic { assume mem[l]=0; mem[l]:=tid; fence; }
  r:=mem[Obj];
  if r=0 then { r:=NewIdx(); mem[Obj]:=r; }
  mem[l]:=0;
 }
}
```

Figure 6. The code for double check initialization. The parameter `a` holds the address for `Obj`.

5.1 Example - Double Checked Initialization

Consider the code given in Fig. 6, which is derived from double checked initialization. The objective of the procedure is to ensure that a shared object *Obj* is initialized exactly once. In order to simplify presentation, we assume that *Obj* fits in a single memory slot, addressed by `mem`[`Obj`].

The procedure starts by checking whether the object of interest *Obj* has already been initialized. If *Obj* has indeed been initialized, the procedure terminates. Otherwise, the procedure switches to the initialization phase. This phase starts by acquiring a (global) lock which protects the initialization operation on the object. If the lock is available, which is when `mem`[l] is equal to 0, it is acquired by setting `mem`[l] to the thread identifier of the current thread, `tid`, and flushing the contents of the buffer ensuring the global visibility of the lock being acquired. Interestingly enough, this is the only place in the code where a fence is used.

After successfully acquiring the lock, the current state of *Obj* is checked again. This is necessary to ensure that no other (concurrent thread) has initialized *Obj* in the meantime. If *Obj* is initialized by some other thread, the lock is released by resetting `mem`[l] back to 0 and the procedure terminates. Otherwise, a new memory slot is prepared and *Obj* is assigned to this new slot, which is followed by the release of the lock and the termination of the procedure.

Triangular race. This code has a triangular race due to the first read of Obj, but before getting into that let us first discuss why the read of Obj within the lock protected region does not lead to a triangular race. Let us use R_1 and R_2 to denote the first and second read statements of `mem`[`Obj`]. Let P be the program containing only `DoubleCheckInit`.

Let $\vec{r}$ be a TSO-run of the form

$$\vec{r} = q_0\tau_1 \cdot \; q \xrightarrow{\textsc{Rd},t:R_2} q' \; \cdot \tau_2 q_n$$

For $\vec{r}$ to contain a triangular race, one condition requires that t have done a write on some variable other than that is read by R_2, i.e. a local write statement to a location different from `Obj` in τ_1. Such a write, the one that updates `mem`[l] in order to acquire the lock, exists. The second condition requires that another thread write concurrently to `mem`[`Obj`] as the first transition in τ_2. But clearly such a concurrent write cannot occur because all writes to `mem`[`Obj`] are protected by the lock and at the beginning of τ_2, the thread t holds the lock. Thus, had it not been for the first read R_1, this procedure would have been triangular race-free and SC-like.

Now let us repeat the same consideration for R_2. Let the TSO-run $\vec{r}$ be in the form

$$\vec{r} = q_0\tau_1 \cdot \; q \xrightarrow{\textsc{Wr},t:W_l} \cdot \, \tau_2 \; \cdot q' \rightarrow$$

6. Speaking Locally - QED and TSO

7. Reduction for Other Relaxed Memory Models

8. Conclusion

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...