Using coarse-grained abstractions to verify linearizability on TSO architectures

John Derrick[1], Graeme Smith[2], Lindsay Groves[3], and Brijesh Dongol[1]

[1]Department of Computing, University of Sheffield, UK
[2]School of Information Technology and Electrical Engineering, The University of Queensland, Australia
[3] School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

Abstract. Most approaches to verifying linearizability assume a sequentially consistent memory model, which is not always realised in practice. In this paper we study correctness on a *weak* memory model: the TSO (Total Store Order) memory model, which is implemented in x86 multicore architectures.
Our central result is a proof method that simplifies proofs of linearizability on TSO. This is necessary since the use of local buffers in TSO adds considerably to the verification overhead on top of the already subtle linearizability proofs. The proof method involves constructing a coarse-grained abstraction as an intermediate layer between an abstract description and the concurrent algorithm. This allows the linearizability proof to be split into two smaller components, where the effect of the local buffers in TSO is dealt with at a higher level of abstraction than it would have been otherwise.

1 Introduction

There has been extensive work on correctness of fine-grained concurrent algorithms over the last few years, where linearizability is the key criteria that is applied. This requires that fine-grained implementations of access operations (e.g., insertion or removal of an element of a data structure) appear as though they take effect "instantaneously at some point in time" [12], thereby achieving the same effect as an atomic operation. There has been considerable work on verifying linearizability, and a variety of proof techniques have been developed, some of them with automated support.

However, most of this work assumes a particular memory model; specifically a *sequentially consistent* (SC) memory model, whereby program instructions are executed by the hardware in the order specified by the program. Typical multicore systems communicate via shared memory and, to increase efficiency, use (local) store buffers. Whilst these *relaxed memory models* give greater scope for optimisation, sequential consistency is lost, and because memory accesses may be reordered in various ways it is even harder to reason about correctness. Typical multiprocessors that provide such weaker memory models include the x86 [16], Power [17] and ARM [1] multicore processor architectures.

In this paper we focus on one such memory model, the TSO (Total Store Order) model [17] which is implemented in the x86 architecture. The notion of correctness we adopt for this architecture is TSO-linearizability as defined in [9]. If verifying linearizability was not hard enough, the reordering of the memory accesses in TSO brings an additional layer of complexity. The purpose of this paper is to simplify this complexity as much as we can. To do so we use the key observation that in many cases for an algorithm on TSO the conditions that linearizability require can be split into two. One aspect deals with the fine-grained nature of the concurrent algorithm, and the other with the effect the local buffers have on when effects become visible in the shared memory.

We exploit this in our proof method, which uses an intermediate description, specifically a coarse-grained abstraction that lies between the abstract specification and the concurrent algorithm. The coarse-grained abstraction captures the semantics of the concurrent algorithm when there is no fine-grained interleaving of operations by different processes. Our simplified proof method then requires one set of proof obligations between the concurrent algorithm and the coarse-grained abstraction, and a different set of proof obligations between the coarse-grained abstraction and the abstract description.

The structure of the paper is as follows. In Section 2 we introduce the TSO model as well as our running example, the *spinlock* algorithm along with an abstract and concrete specification of it in Z. (We assume the reader is familiar with Z — for details see [18]). In Section 3 we provide a coarse-grained abstraction of spinlock. In Section 4 we adapt the standard definition of linearizability to allow the concrete specification to be proved linearizable to the coarse-grained specification. In Section 5 we define a transformation from the coarse-grained abstraction to the abstract one which together with the results of Section 4 allows us to prove overall correctness of the concrete specification with the abstract one. This is shown to be sound in Section 6 with respect to a notion of linearizability on TSO previously published in [9]. We conclude in Section 7.

2 The TSO memory model

In the TSO architecture [17] each processor core uses a write buffer, which is a FIFO queue that stores pending writes to memory. A processor core performing a *write* to a memory location enqueues the write to the buffer and continues computation without waiting for the write to be committed to memory. Pending writes do not become visible to other cores until the buffer is *flushed*, which commits (some or all) pending writes to memory. The value of a memory location *read* by a process is the most recent in the processor's local buffer. If there is no such value (e.g., initially or when all writes corresponding to the location have been flushed), the value of the location is fetched from memory. The use of local buffers allows a read by one process, occurring after a write by another, to return an older value as if it occurred before the write.

In general, flushes are controlled by the CPU, and from the programmer's perspective occur non-deterministically. However, a programmer may explicitly include a *fence*, or *memory barrier*, instruction in a program's code to force a flush to occur. Therefore, although TSO allows some non-sequentially consistent executions, it is used in many modern architectures on the basis that these can be prevented, where necessary, by programmers using fence instructions. A pair of *lock* and *unlock* commands allows a process to acquire sole access to the memory. Both commands include a fence which forces the store buffer of that process to be flushed completely.

2.1 Example – spinlock

Spinlock is a locking mechanism designed to avoid operating system overhead associated with process scheduling and context switching. The **abstract** specification simply describes a lock, with operations $Acquire_p$, $Release_p$ and $TryAcquire_p$ parameterised by the identifier of the process $p \in P$ performing the operation (P is the set of all process identifiers). A global variable x represents the lock and is set to 0 when the lock is held by a thread, and 1 otherwise. As in [16], we assume that only a process that has acquired the lock will release it, and a process will only attempt to acquire the lock if it doesn't already hold it.

AS

$x : \{0, 1\}$

Init

AS

$x = 1$

Acquire$_p$

ΔAS

$x = 1$

$x' = 0$

Release$_p$

ΔAS

$x' = 1$

TryAcquire$_p$

ΔAS

$out! : \{0, 1\}$

---

if $x = 1$

then $x' = 0 \wedge out! = 1$

else $x' = x \wedge out! = 0$

A typical implementation of spinlock [11] is shown in Figure 1, given as pseudo-code (where `a1`, etc. are line numbers). A thread trying to acquire the lock *spins*, i.e., waits in a loop, while repeatedly checking `x` for availability.

```
  word x=1;

  void acquire()          void release()          int tryacquire()
    {                       {                       {
a1   while(1) {         r1    x=1;             t1    lock;
a2      lock;               }                  t2    if (x==1) {
a3      if (x==1) {                            t3       x=0;
a4         x=0;                                t4       unlock;
a5         unlock;                             t5       return 1;
a6         return;                                    }
        }                                      t6    unlock;
a7      unlock;                                t7    return 0;
a8      while(x==0){};                              }
     }}
```

Fig. 1. Spinlock implementation

A terminating `acquire` operation will always succeed to acquire the lock. It will lock[1] the global memory so that no other process can write to `x`. If, however, another

[1] Locking the global memory using the TSO `lock` command should not be confused with acquiring the lock of this case study by setting `x` to 0.

thread has already acquired the lock (i.e., `x==0`) then it will unlock the global memory and spin, i.e., loop in the while-loop until it becomes free, before starting over. Otherwise, it acquires the lock by setting `x` to 0.

The operation `release` releases the lock by setting `x` to 1. The `tryacquire` operation differs from `acquire` in that it only makes one attempt to acquire the lock. If this attempt is successful it returns 1, otherwise it returns 0.

The `lock` and `unlock` commands act as memory barriers on TSO. Hence, writes to `x` by the `acquire` and `tryacquire` operations are not delayed. For efficiency, however, `release` does not have a memory barrier and so its write to `x` can be delayed until a flush occurs. The spinlock implementation will still work correctly, the only effect that the absence of a barrier has is that a subsequent `acquire` may be delayed until a `flush` occurs, or a `tryacquire` operation by a thread q may return 0 after the lock has been released by another thread p. For example, the following execution is possible, where we write $(q, \texttt{tryacquire}(0))$ to denote process q performing a `tryacquire` operation and returning 0, and $\texttt{flush}(p)$ to denote the CPU flushing a value from process p's buffer: $\langle (p, \texttt{acquire}), (p, \texttt{release}), (q, \texttt{tryacquire}(0)), \texttt{flush}(p) \rangle$.

Thus p performs an `acquire`, then a `release`, and then q a `tryacquire` that returns 0 even though it occurs immediately after the `release`. This is because the $\texttt{flush}(p)$, which sets the value of x in memory to 0 has not yet occurred.

The Z specification that corresponds to the **concrete** system has one operation per line of pseudo-code, and each operation can be invoked by a given process. The concrete state consists of the shared memory, given as a global state *GS* and local state *LS* for each process. *GS* includes the value of the shared variable x (initially 1), a variable *lock* which has value $\{p\}$ when a process p currently has the global memory locked (and is $\varnothing$ otherwise), and a buffer for each process modelled as a sequence of 0 and 1's.[2]

$$\begin{array}{|l}
\hline \textit{GS} \\
x : \{0,1\} \\
\textit{lock} : \mathbb{P}\, P \\
\textit{buffer} : P \to \mathrm{seq}\{0,1\} \\
\hline
\#\textit{lock} \leq 1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \textit{GSInit} \\
\textit{GS} \\
\hline
x = 1 \\
\textit{lock} = \varnothing \\
\forall\, p : P \bullet \textit{buffer}(p) = \langle\, \rangle \\
\hline
\end{array}$$

For a given process, *LS* is specified in terms a program counter, *PC*, indicating which operations (i.e., lines of code) can next be performed. Let

$$PC \ ::= \ 1 \mid a1 \mid \ldots \mid a8 \mid t1 \mid \ldots \mid t7 \mid r1$$

The value 1 denotes that the process is not executing any of the three operations. The values ai, for $i \in 1 \mathrel{..} 8$, denote the process is ready to perform the ith line of code of `acquire`, and similarly for ti and `tryacquire`. The value $r1$ denotes the process is ready to perform the first line of `release`.

$$\begin{array}{|l}
\hline \textit{LS} \\
pc : PC \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \textit{LSInit} \\
\textit{LS} \\
\hline
pc = 1 \\
\hline
\end{array}$$

[2] In a more complex example, the buffer would also store the name of the variable assigned.

Given this specification, the lines of code are formalised as Z operations.[3] For a given process p, we have an operation $A0_p$ corresponding to the invocation of the `acquire` operation, and an operation $A1_p$ corresponding to the line of code `while(1)`.

$$\begin{array}{|l} \hline A0_p \\ \Xi GS;\ \Delta LS \\ \hline pc = 1 \wedge pc' = a1 \\ \hline \end{array} \qquad \begin{array}{|l} \hline A1_p \\ \Xi GS;\ \Delta LS \\ \hline pc = a1 \wedge pc' = a2 \\ \hline \end{array}$$

The operation $A2_p$ corresponds to the line of code `lock`. To model the next line of code, `if (x==1)`, we use two operations: $A31_p$ for the case when $x = 1$, and $A30_p$ for the case when $x = 0$. These operations are only enabled when the buffer is empty, modelling the fact that the lock of $A2_p$ is a fence, i.e., a sequence of flush operations on p's buffer (specified below) must occur immediately after $A2_p$ if the buffer is non-empty.

$$\begin{array}{|l} \hline A2_p \\ \Delta GS;\ \Delta LS \\ \hline pc = a2 \wedge lock = \varnothing \\ pc' = a3 \wedge lock' = \{p\} \\ \hline \end{array} \qquad \begin{array}{|l} \hline A31_p \\ \Xi GS;\ \Delta LS \\ \hline buffer(p) = \langle\,\rangle \\ pc = a3 \wedge x = 1 \\ pc' = a4 \\ \hline \end{array} \qquad \begin{array}{|l} \hline A30_p \\ \Xi GS;\ \Delta LS \\ \hline buffer(p) = \langle\,\rangle \\ pc = a3 \wedge x = 0 \\ pc' = a7 \\ \hline \end{array}$$

The operation $A4_p$, corresponding to the line `x=0`, adds the value 0 to the buffer. The operations corresponding to the other lines of `acquire` are modelled similarly. The two operations corresponding to `while(x==0)`, $A80_p$ and $A81_p$, are only enabled when either x can be read from the buffer, i.e., $buffer \neq \langle\,\rangle$, or the buffer is empty and the memory is not locked (and so x can be read from the global memory).

$$\begin{array}{|l} \hline A4_p \\ \Delta GS;\ \Delta LS \\ \hline pc = a4 \\ buffer'(p) = buffer(p) \frown \langle 0 \rangle \\ pc' = a5 \\ \hline \end{array} \qquad \begin{array}{|l} \hline A5_p \\ \Delta GS;\ \Delta LS \\ \hline buffer(p) = \langle\,\rangle \\ pc = a5 \wedge pc' = a6 \wedge lock' = \varnothing \\ \hline \end{array}$$

$$\begin{array}{|l} \hline A6_p \\ \Xi GS;\ \Delta LS \\ \hline pc = a6 \wedge pc' = 1 \\ \hline \end{array} \qquad \begin{array}{|l} \hline A7_p \\ \Delta GS;\ \Delta LS \\ \hline buffer(p) = \langle\,\rangle \\ pc = a7 \wedge pc' = a8 \wedge lock' = \varnothing \\ \hline \end{array}$$

$$\begin{array}{|l} \hline A80_p \\ \Xi GS;\ \Xi LS \\ \hline pc = a8 \\ buffer(p) = \langle\,\rangle \Rightarrow lock = \varnothing \wedge x = 0 \\ buffer(p) \neq \langle\,\rangle \Rightarrow last\,buffer(p) = 0 \\ \hline \end{array} \qquad \begin{array}{|l} \hline A81_p \\ \Xi GS;\ \Delta LS \\ \hline pc = a8 \\ buffer(p) = \langle\,\rangle \Rightarrow lock = \varnothing \wedge x = 1 \\ buffer(p) \neq \langle\,\rangle \Rightarrow last\,buffer(p) = 1 \\ pc' = a1 \\ \hline \end{array}$$

[3] To simplify the presentation we adopt the convention that the values (of variables or in the range of a function) that are not explicitly changed by an operation remain unchanged.

The operations for `tryacquire` are similar to those of `acquire`. Those for `release` are given below. We also have an operation, Flush_{cpu}, corresponding to a CPU-controlled flush which outputs the process whose buffer it flushes.

$$\begin{array}{|l}
\hline
R0_p \\
\Xi GS \\
\Delta LS \\
\hline
pc = 1 \land pc' = r1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
R1_p \\
\Xi GS \\
\Delta LS \\
\hline
pc = r1 \land pc' = 1 \\
\mathit{buffer}'(p) = \mathit{buffer}(p) \frown \langle 1 \rangle \\
\hline
\end{array}$$

We also have an operation, Flush_{cpu}, corresponding to a CPU-controlled flush which outputs the process whose buffer it flushes.

$$\begin{array}{|l}
\hline
\mathit{Flush}_{cpu} \\
\Delta GS \\
p! : P \\
\hline
\mathit{lock} = \varnothing \lor \mathit{lock} = \{p!\} \\
\mathit{buffer}(p!) \neq \langle\,\rangle \Rightarrow x' = \mathit{head}\,\mathit{buffer}(p!) \land \mathit{buffer}'(p!) = \mathit{tail}\,\mathit{buffer}(p!) \\
\mathit{buffer}(p!) = \langle\,\rangle \Rightarrow x' = x \land \mathit{buffer}'(p!) = \mathit{buffer}(p!) \\
\hline
\end{array}$$

The task in its most general setting is to prove that this concrete specification is linearizable with respect to the abstract one given earlier. The rest of this paper is concerned with a method by which one can show this and similar algorithms correct. First we recap on the notion of linearizability and then discuss how it can be used to provide a coarse-grained abstraction of our concrete specification.

3 Coarse-grained abstraction

Linearizability [12] is the standard notion of correctness for concurrent algorithms, and allows one to compare a fine-grained implementation against its abstract specification. For example, in spinlock the concurrent system might perform an execution such as: $\langle (p, \mathtt{A0}), (q, \mathtt{R0}), (p, \mathtt{A1}), (q, \mathtt{R1}) \rangle$. The idea of linearizability is that any such concrete sequence must be consistent with *some* abstract execution (i.e., a sequence of *Acquire*'s, *Release*'s etc. also performed by p and q):

> (1) Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its return. This point is known as the *linearization point* [12].

In other words, if two operations overlap, then they may take effect in any order from an abstract perspective, but otherwise they must take effect in program order.

There has been an enormous amount of interest in deriving techniques for verifying linearizability. These range from using shape analysis [2, 4] and separation logic [4] to rely-guarantee reasoning [20] and refinement-based simulation methods [10, 7]. Most of this work has been for sequentially consistent architectures, but some work has been done for TSO [3, 11, 19, 9]. In particular, in [9] we have defined a simulation-based proof method for linearizability on TSO. The key point in defining linearizability on TSO is to take into account the role of the local buffers. Since the flush of a process's buffer is sometimes the point that the effect of an operation's changes to memory become globally visible, the flush can be viewed as being the final part of the operation. For example, the flush of a variable, such as `x`, after an operation, such as `release`, can be taken as the return of that operation. Under this interpretation, the `release` operation extends from its invocation to the flush which writes its change to `x` to the global memory. Thus [19] and [9] use the following principle:

> (2) The return point of an operation on a TSO architecture is not necessarily the point where the operation ceases execution, but can be any point up to the last flush of the variables written by that operation.

However, any proof method will be complicated by having to deal with both the inherent interleaving handled by linearizability and the additional potential overlapping of concrete operations resulting from the above principle. For example, in spinlock, a process may perform a `release` but not have its buffer flushed before invoking its next operation.

The idea in this paper is simple. We use an intermediate specification (between the abstract and concrete) to split the original proof obligations into two simpler components. The first (between the concrete and intermediate specifications) deals with the underlying linearizability, and the second (between intermediate and abstract) deals with the effects of local buffers. The intermediate specification is a *coarse-grained abstraction* that captures the semantics of the concrete specification with no fine-grained interleaving of operations by different processes. We describe how to define such a coarse-grained abstraction in the next section.

Figure 2 illustrates this idea for a specific execution: at the bottom is a concrete execution, and in the middle is an execution of the intermediate specification which linearizes it (as per Section 4). Finally at the top is an execution of the abstract specification that is related to the intermediate one by the transformation *TRANS* defined in Section 5. Overall this will guarantee that the concrete execution is TSO-linearizable to the abstract one, as we show in Section 6.

3.1 Defining the coarse-grained abstraction

The coarse-grained abstraction is constructed by adding local buffers to the abstract specification. Thus, it is still a description on the TSO architecture – since it has buffers and flushes – but does not decompose the operations. The state space is the abstract state space with the addition of a buffer for each process (as in the concrete state space *GS*). Like in the concrete state space, all buffers are initially empty. Hence for spinlock we have:

BS

$x : \{0, 1\}$

$buffer : P \rightarrow \mathrm{seq}\{0, 1\}$

BSInit

BS

$x = 1 \wedge \forall p : P \bullet buffer(p) = \langle\,\rangle$

Each operation is like that of the abstract specification except that

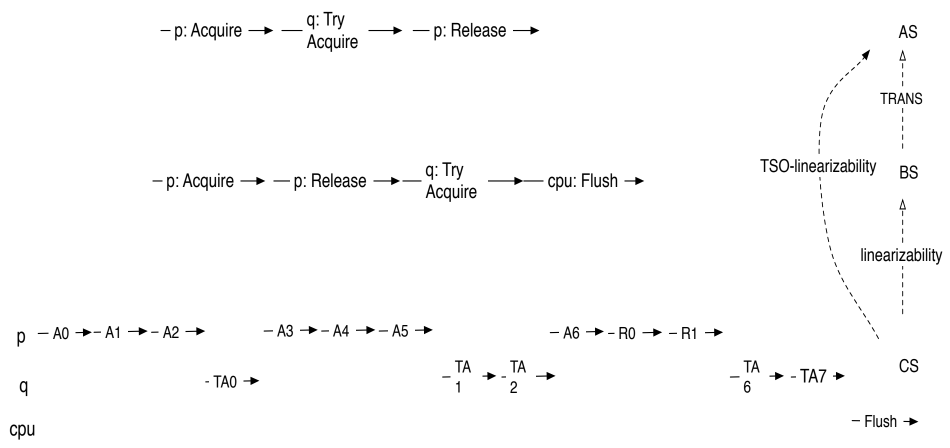

Fig. 2. Three executions in abstract, intermediate and concrete models

- reads are replaced by reads from the process's buffer or from memory, i.e., the operation refers to the latest values of variables in the buffer, and to their actual values otherwise,
- writes are replaced by writes to the buffer (unless the corresponding concrete operation has a fence),
- because we have buffers in the intermediate state space we need to include fences and flushes: the buffer is set to empty when the corresponding concrete operation has a fence, and a flush is modelled as a separate operation.

For example, for the abstract operation $Acquire_p$, $x = 1$ represents a read, and $x' = 0$ represents a write. Using the above heuristic, we replace $x = 1$ by $buffer(p) \neq \langle\,\rangle \Rightarrow last\,buffer(p) = 1 \wedge buffer(p) = \langle\,\rangle \Rightarrow x = 1$ since the latest value of x is that in the buffer when the buffer is not empty, and the actual value of x otherwise. We also replace $x' = 0$ by $buffer'(p) = \langle\,\rangle \wedge x' = 0$ since the corresponding concrete operation has a fence. Similarly, while the operation $TryAcquire_p$ writes directly to x and sets the buffer to empty (since it has a fence), the operation $Release_p$ writes only to the buffer.

$Acquire_p$

ΔBS

$buffer(p) \neq \langle\,\rangle \Rightarrow last\,buffer(p) = 1$
$buffer(p) = \langle\,\rangle \Rightarrow x = 1$
$buffer'(p) = \langle\,\rangle \wedge x' = 0$

$Release_p$

ΔBS

$buffer'(p) = buffer(p) \frown \langle 1 \rangle$

$TryAcquire_p$

ΔBS
$out! : \{0, 1\}$

if $buffer(p) \neq \langle\,\rangle \wedge last\,buffer(p) = 1 \vee buffer(p) = \langle\,\rangle \wedge x = 1$
then $buffer'(p) = \langle\,\rangle \wedge x' = 0 \wedge out! = 1$
else $buffer'(p) = \langle\,\rangle \wedge x' = 0 \wedge out! = 0$

Note that $x' = 0$ in the else-predicate of $TryAcquire_P$ since if the buffer is empty, x is 0 and does not change, and if the buffer is not empty, the last element in buffer is 0 and the buffer is completely flushed by the `lock` command in `tryacquire`.

Finally, the course-grained abstraction is completed with the $Flush_{cpu}$ operation. As in the concrete specification, this operation is performed by the CPU process.

$$\begin{array}{|l}
\multicolumn{1}{l}{Flush_{cpu}} \\
\Delta BS \\
p! : P \\
\hline
buffer(p!) \neq \langle\,\rangle \Rightarrow x' = head\, buffer(p!) \wedge buffer'(p!) = tail\, buffer(p!) \\
buffer(p!) = \langle\,\rangle \Rightarrow x' = x \wedge buffer'(p!) = buffer(p!) \\
\hline
\end{array}$$

The coarse-grained abstraction is chosen purposefully to reflect the abstract specification; this facilitates the final part of the proof. The inclusion of buffers and flush operations, however, means it can be shown to linearize the concrete specification using standard proof methods.

4 Linearizability: From concrete to intermediate specification

To prove the concrete specification is correct with respect to the intermediate one, we can use a slight adaption of the standard notion of linearizability. Below we describe how we adapt the formal definition and proof method for linearizability given in [7].

In the standard definition of linearizability, *histories* are sequences of *events* which can be invocations or returns of operations from a set I and performed by a particular process from a set P. On the TSO architecture, operations can be flushes and we assume that a flush is only executed by a CPU process $cpu \in P$, different from all other processes. We also assume that invocations of flushes are immediately followed by their returns. Invocations have an associated input from domain In, and returns an output from domain Out.

$$\begin{array}{l}
Event ::= inv\langle\!\langle P \times I \times In \rangle\!\rangle \mid ret\langle\!\langle P \times I \times Out \rangle\!\rangle \\
History == \mathrm{seq}\, Event
\end{array}$$

For a history h, $\#h$ is the *length* of the sequence, and $h(n)$ its nth element (for $n : 1..\#h$). Predicates $inv?(e)$ and $ret?(e)$ determine whether an event $e \in Event$ is an invoke or return, respectively. We let $e.\pi \in P$ and $e.i \in I$ be the process executing the event e and the operation to which the event belongs, respectively.

Let $mp(p, m, n, h)$ denote matching pairs of invocations and returns by process p in history h as in [7]. Its definition requires that $h(m)$ and $h(n)$ are executed by the same process p and are an invocation and return event, respectively, of the same operation. Additionally, it requires that for all k between m and n, $h(k)$ is not an invocation or return event of p. That is, $mp(p, m, n, h)$ holds iff

$$\begin{array}{l}
0 < m < n \leq \#h \wedge \\
inv?(h(m)) \wedge ret?(h(n)) \wedge h(m).\pi = h(n).\pi = p \wedge h(m).i = h(n).i \wedge \\
\forall k \bullet m < k < n \Rightarrow h(k).\pi \neq p
\end{array}$$

We say a history h is *legal* iff for each $n : 1..\#h$ such that $ret?(h(n))$, there exists an earlier $m : 1..n-1$ such that $mp(p,m,n,h)$.

A formal definition of linearizability is given below. A history is *incomplete* if it has either (i) an operation which has been invoked and has linearized but not yet returned, or (ii) results in a non-empty buffer. An incomplete history h is extended with a sequence h_0 of flushes and returns of non-flush operations, then matched to a sequential history hs by removing the remaining pending invocations using a function *complete*. Let $Hist_{FR}$ be the set of histories that are sequences of flushes and returns of non-flush operations.

Definition 1 (Linearizability). *A history $h : History$ is* linearizable *with respect to some sequential history hs iff $lin(h,hs)$ holds, where*

$$lin(h,hs) \mathrel{\widehat{=}} \exists\, h_0 : Hist_{FR} \bullet legal(h \frown h_0) \wedge linrel(complete(h \frown h_0), hs)$$

where

$$\begin{aligned} linrel(h,hs) \mathrel{\widehat{=}} \exists f : 1..\#h \rightarrowtail\!\!\!\!\twoheadrightarrow 1..\#hs \bullet (\forall\, n : 1..\#h \bullet h(n) = hs(f(n))) \wedge \\ (\forall\, p : P;\ m,n : 1..\#h \bullet m < n \wedge mp(p,m,n,h) \Rightarrow f(n) = f(m)+1) \wedge \\ (\forall\, p,q : P;\ m,n,m',n' : 1..\#h \bullet \\ n < m' \wedge mp(p,m,n,h) \wedge mp(q,m',n',h) \Rightarrow f(n) < f(m')) \quad \Box \end{aligned}$$

That is, operations in *hs* do not overlap (each invocation is followed immediately by its matching return) and the order of non-overlapping operations in h is preserved in *hs*.

For example, the history h corresponding to the concrete execution in Figure 2 is

$\langle inv(p, \texttt{acquire},), inv(q, \texttt{tryacquire},), ret(p, \texttt{acquire},), inv(p, \texttt{release},),$
$ret(p, \texttt{release},), ret(q, \texttt{tryacquire}, 0), inv(cpu, \texttt{flush},), ret(cpu, \texttt{flush}, p)\rangle$

This history is complete and legal, and is linearized by the history *hs*

$\langle inv(p, Acquire,), ret(p, Acquire,), inv(p, Release,), ret(p, Release,),$
$inv(q, TryAcquire,), ret(q, TryAcquire, 0), inv(cpu, Flush,), ret(cpu, Flush, p)\rangle$

which corresponds to the intermediate-level execution in Figure 2.

Correctness requires showing all concrete histories are linearizable. Existing proof methods for showing this include the simulation-based approach in [7]. This is based on showing that the concrete specification is a non-atomic refinement of the abstract one. Examples of its use are given in [5–8, 14, 15]. This approach is fully encoded in a theorem proving tool, KIV [13], and has been proved sound and complete — the proofs themselves being done within KIV. The key point for us is that, for this portion of the correctness proof, we do not have to adapt the proof method.

5 Transforming the intermediate specification to an abstract one

The previous section has shown how to prove that a concrete specification is linearizable with respect to an intermediate, coarse-grained abstraction. The inclusion of local buffers in this intermediate specification avoided us needing to deal with the effects of the TSO architecture. In this section, we introduce a deterministic history transformation which when coupled with the linearization method of the previous section guarantees the overall correctness of concrete specification with respect to the abstract one. Correctness involves showing every history of the intermediate specification is transformed to a history of the abstract one. Soundness of this approach is proved in Section 6.

The histories of the intermediate specification are sequential, i.e., returns of operations occur immediately after their invocations, but the specification includes buffers and flush operations. The transformation turns the histories of the intermediate specification into histories of an abstract one, i.e., without buffers, with the same behaviour. It does this according to principle (2) in Section 3, i.e., it moves the return of an operation to the flush that make its global behaviour visible. To keep histories sequential, we also move the invocation of the operation to immediately before the return.

To define the transformation, denoted *TRANS*, we need to calculate which flush an operation's return (and invocation) should be moved to. This is done by a function *mpf* (standing for *matching pair flush*) which in turn uses *mp* defined in Section 4. A flush returns an operation, i.e., makes its effects visible globally, when it writes the last variable which was updated by that operation to memory. Let $bs(p,m,h)$ denote the size of process p's buffer at point m in the history h. Given an operation whose invocation is at point m and return at point n, if the buffer is empty when the operation is invoked, then the number of flushes to be performed before the operation returns is equal to the size of the buffer at the end of the operation, i.e., $bs(p,n,h)$; if this number is 0 then the return does not move. Similarly, if an operation contains a fence then the number of flushes before the operation returns is also equal to $bs(p,n,h)$. In all other cases, we need to determine whether the operation has written to any global variables. If it has written to one or more global variables then again the number of flushes to be performed before the operation returns is $bs(p,n,h)$.

To determine whether an operation has written to global variables, we compare the size of the buffer at the start and end of the operation taking into account any flushes that have occurred in between. Let $nf(p,m,n,h)$ denote the number of flushes of process p's buffer from point m up to and including point n in h. The number of writes between the two points is given by

$$nw(p,m,n,h) \mathrel{\widehat{=}} bs(p,n,h) - bs(p,m,h) + nf(p,m,n,h) \ .$$

The function *mpf* is then defined below where m, n and l are indices in h such that (m,n) is a matching pair and l corresponds to the point to which the return of the matching pair must be moved.

$$\begin{array}{l} mpf(p,m,n,l,h) \mathrel{\widehat{=}} mp(p,m,n,h) \wedge n \leq l \wedge \\ \qquad\qquad\qquad \mathsf{if}\ nw(p,m,n,h) = 0 \vee bs(p,n,h) = 0\ \mathsf{then}\ l = n \\ \qquad\qquad\qquad \mathsf{else}\ h(l) = ret(cpu, Flush, p) \wedge nf(p,n,l,h) = bs(p,n,h) \end{array}$$

The first part of the if states that $l = n$ if no items are put on the buffer by the operation invoked at point m, or all items put on the buffer have already been flushed when the operation returns. The second states that l corresponds to a flush of p's buffer and the number of flushes between n and l is precisely the number required to flush the contents of the buffer at n.

The history transformation *TRANS* is then defined as follows. It relies on the fact that the intermediate histories are sequential, i.e., comprise a sequence of matching pairs. Each matching pair of a history is either moved to the position of the flush which acts as its return (given by *mpf*), or left in the same position relative to the other matching pairs. The transformation also removes all flushes from the history. Informally we can think of $TRANS(hs)$ creating a new history determined by applying two steps to the history hs. The first step introduces a new history hs_1 which includes dummy events δ and invocations and returns of flushes. The second step removes these resulting in an abstract history:

Step 1. For all indices m, n and l such that $mpf(p, m, n, l, h)$ holds for some p:

```
    if $n = l$ then $hs_1(m) := hs(m)$ and $hs_1(n) := hs(n)$
    else $hs_1(l) := hs(n)$ and $hs_1(l-1) := hs(m)$ and $hs_1(n) := \delta$ and $hs_1(m) := \delta$
```

Step 2. All δ and flush invocations and returns are removed.

Although this is the best intuition of *TRANS*, the formal definition is based on identifying the matching pairs, and ordering them by the positions that invocations and returns are moved to. The key point is that the positions that returns get moved to are different for each event, so we can order them, and this order defines our new history.

Definition 2 (TRANS). *Let hs be a history of the intermediate specification,* $S = \{(m, n, l) \mid \exists p : P \bullet mpf(p, m, n, l, hs) \wedge hs(m).i \neq Flush\}$*, and* $k = \#S$*. We can order elements of S by the 3rd element in the tuple:* $l_1 < l_2 < \ldots < l_k$*. Then* $TRANS(hs)$ *is an abstract history with length* $2k$ *defined (for* $i : 1 \mathinner{..} 2k$*) as:*

$$TRANS(hs)(i) = \begin{cases} hs(n) & \text{if } i \text{ is even and } (m, n, l_{i/2}) \in S \\ hs(m) & \text{if } i \text{ is odd and } (m, n, l_{(i+1)/2}) \in S \end{cases}$$

Furthermore, this mapping induces a function G which identifies the index that any particular invocation or return has been moved to. G is defined (for $j : 1 \mathinner{..} \#hs$*) by:*

$$G(j) = \begin{cases} 2i & \text{if } (m, j, l_i) \in S \text{ and so } hs(j) \text{ is a return} \\ 2i - 1 & \text{if } (j, n, l_i) \in S \text{ and so } hs(j) \text{ is an invocation} \end{cases}$$ □

Definition 3 (TSO-equivalence). *An intermediate specification BS is TSO-equivalent to an abstract specification AS whenever for every history hs of BS,* $TRANS(hs)$ *is a history of AS.* □

For example, given the intermediate-level history hs in Section 4, the indices which are related by *mpf* are as follows: for *Acquire* we get $mpf(p, 1, 2, 2, hs)$, for *Release* we get $mpf(p, 3, 4, 8, hs)$, for *TryAcquire* we get $mpf(q, 5, 6, 6, hs)$ and for *Flush* we get $mpf(cpu, 7, 8, 8, hs)$. S will include the first three tuples which are then ordered: $(1, 2, l_1)$, $(5, 6, l_2)$, $(3, 4, l_3)$ (where $l_1 = 2$, $l_2 = 6$ and $l_3 = 8$). Thus, $TRANS(hs)(1) = hs(1)$ since 1 is odd and $(1, 2, l_1) \in S$. Similarly, $TRANS(hs)(6) = hs(4)$ as 6 is even and $(3, 4, l_3) \in S$. Overall, $TRANS(hs)$ is the following which corresponds to the abstract execution in Figure 2: $\langle inv(p, Acquire,), ret(p, Acquire,), inv(q, TryAcquire,), ret(q, TryAcquire, 0), inv(p, Release,), ret(p, Release,)\rangle$.

6 Gluing it together: From concrete to abstract specification

Overall, we want to show the correctness of the concrete specification with respect to the abstract one. The notion of correctness we adopt is TSO-linearizability as defined in [9]. We summarise this definition below before proving that the effect of linearizability followed by TSO-equivalence implies TSO-linearizability.

6.1 TSO-linearizability

To prove linearizability on TSO, we introduce a history transformation *Trans* which (according to principle (2) in Section 3) moves the return of each operation to the flush which makes its global behaviour visible, if any. *Trans* is similar to *TRANS* of Section 5 except it does not also move the invocation of the operation. The informal intuition for *Trans* alters the first step of the transformation to the following:
Step 1. For all indices m, n and l such that $mpf(p,m,n,l,h)$ holds for some p:

```
    if $n = l$ then $h_1(m) := h(m)$ and $h_1(n) := h(n)$
    else $h_1(m) := h(m)$ and $h_1(l) := h(n)$ and $h_1(n) := \delta$
```

In a manner similar to *TRANS*, this is formalised in the following definition:

Definition 4 (Trans). *Let h be a history of the concrete specification,* $S_1 = \{(m,n,l,x) \mid \exists p : P \bullet mpf(p,m,n,l,h) \wedge h(m).i \neq Flush \wedge x \in \{m,l\}\}$*, and* $k_1 = \#S_1$*. We can order the elements of* S_1 *by their 4th elements:* $x_1 < x_2 < \ldots < x_{k_1}$*. Then* $Trans(h)$ *is an abstract history with length* k_1 *defined (for* $i : 1 \,..\, k_1$*) as:*

$$Trans(h)(i) = \begin{cases} h(x_i), & \textit{if } (x_i,n,l,x_i) \in S\textit{, for some } n \textit{ and } l \\ h(n), & \textit{if } (m,n,x_i,x_i) \in S\textit{, for some } m \end{cases}$$

Furthermore, this mapping induces a function g which identifies the index that any particular invocation or return has been moved from. g is defined (for $i : 1 \,..\, k_1$*) by:*

$$g(i) = \begin{cases} x_i, & \textit{if } (x_i,n,l,x_i) \in S\textit{, for some } n \textit{ and } l \\ n, & \textit{if } (m,n,x_i,x_i) \in S\textit{, for some } m \end{cases}$$ □

For example, given the concrete history h in Section 4, the indices which are related by *mpf* are as follows: for `acquire` we get $mpf(p,1,3,3,h)$, for `tryacquire` we get $mpf(q,2,6,6,h)$, for `release` we get $mpf(p,4,5,8,h)$ and for `flush` we get $mpf(cpu,7,8,8,h)$. The elements of set S_1 are ordered as follows: $(x_1,3,3,x_1)$, $(x_2,6,6,x_2)$, $(1,3,x_3,x_3)$, $(x_4,5,8,x_4)$, $(2,6,x_5,x_5)$, $(4,5,x_6,x_6)$ (where $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, $x_4 = 4$, $x_5 = 6$ and $x_6 = 8$). Thus, $Trans(h)(1) = h(1)$ since $x_1 = 1$, and $Trans(h)(6) = h(8)$ since $x_6 = 8$. Overall $Trans(h)$ is

$\langle inv(p,Acquire,), inv(q,TryAcquire,), ret(p,Acquire,), inv(p,Release,),$
$ret(q,TryAcquire,0), ret(p,Release,)\rangle$.

A key part of adapting the definition of linearizability from Section 4 to TSO is what we mean by a matching pair of invocations and returns. The formal definition of the function *mp* requires that for all k between m and n, $h(k)$ is not an invocation or return event of p. This is not true for our transformed histories on TSO since operations by the same process may overlap. Therefore, we will use a new version of matching pairs mp_{TSO} defined as follows.

$$\begin{array}{l} mp_{TSO}(p,m,n,h) \text{ iff } mpf(p,x,z,y,h) \\ \text{where } m = x - \sum_{p:P} nf(p,1,x,h) \text{ and } n = y - \sum_{p:P} nf(p,1,y,h) \text{ and } x < z \leq y \end{array}$$

We then adopt the definition of TSO-linearizability from [9]. After extending an incomplete concrete history with flushes and returns of non-flush operations, we apply *Trans* to it before matching it to an abstract history.

Definition 5 (TSO-linearizability). *A history h : History is* TSO-linearizable *with respect to some sequential history hs iff $lin_{TSO}(h,hs)$ holds, where*

$$lin_{TSO}(h,hs) \mathrel{\widehat=} \exists\, h_0 : Hist_{FR} \bullet legal(h \frown h_0) \wedge linrel_{TSO}(Trans(complete(h \frown h_0)), hs)$$

where

$$\begin{array}{l} linrel_{TSO}(h,hs) \mathrel{\widehat=} \exists f : 1..\#h \rightarrowtail\!\!\!\!\rightarrow 1..\#hs \bullet (\forall\, n : 1..\#h \bullet h(n) = hs(f(n))) \wedge \\ \qquad (\forall\, p : P;\ m,n : 1..\#h \bullet m < n \wedge mp_{TSO}(p,m,n,h) \Rightarrow f(n) = f(m)+1) \wedge \\ \qquad (\forall\, p,q : P;\ m,n,m',n' : 1..\#h \bullet \\ \qquad\qquad n < m' \wedge mp_{TSO}(p,m,n,h) \wedge mp_{TSO}(q,m',n',h) \Rightarrow f(n) < f(m')) \end{array}$$

We say that a concrete specification is TSO-linearizable with respect to an abstract specification if and only if for all concrete histories h, there exists an abstract history hs such that $lin_{TSO}(h,hs)$. □

The new matching pairs in the example history $Trans(h)$ above are $mp_{TSO}(p,1,3,h_1)$, $mp_{TSO}(q,2,5,h_1)$ and $mp_{TSO}(p,4,6,h_1)$. It is easy to see that this is linearized by the abstract history corresponding to the execution in Figure 2.

6.2 Soundness

Assume a concrete specification *CS* is linearizable with respect to an intermediate specification *BS*, and *BS* is TSO-equivalent to an abstract specification *AS*. Given a concrete history h, to prove our approach sound we have to find an abstract history hs such that $lin_{TSO}(h,hs)$. It is clear that any incomplete concrete history can be extended to a complete and legal history, therefore we assume h is complete and legal.

Since *CS* is linearizable with respect to *BS*, there exists an hs_1 such that $lin(h,hs_1)$ and an associated bijection f_1. Let $hs \mathrel{\widehat=} TRANS(hs_1)$. To show *CS* is TSO-linearizable with respect to *AS*, we define a bijection f between the indices of $Trans(h)$ and hs as follows. Let $f(n) = G(f_1(g(n)))$ where $n \in 1..\#Trans(h)$, and G and g are given in Definitions 2 and 4 respectively. f is a bijection since:

(i) Since $\#h = \#hs_1$ (property of *lin*), we get $\#Trans(h) = \#TRANS(hs_1)$ (since both remove flush invocation and returns) and hence $\#hs = \#Trans(h)$.

(ii) f is surjective since each event in $\mathit{TRANS}(hs_1)$ is either an invocation or return of a non-flush operation. Therefore, there exists an invocation or return of a non-flush operation in hs_1 that is mapped to this event by G. Then surjectivity of f_1 implies there exists an invocation or return of a non-flush operation in h which maps to the event in hs_1. Since this event is of a non-flush operation, there exists an invocation or return in $\mathit{Trans}(h)$ which is mapped to it by g.

(iii) f is injective since g, f_1 and G are all injective.

We now show that f satisfies the three conjuncts of $\mathit{linrel}_{\mathit{TSO}}$ and hence that TSO-linearizability holds.

(i) $\mathit{Trans}(h)(n) = hs(f(n))$ follows by construction of f.

(ii) Given $m, n : 1 \,..\, \#\mathit{Trans}(h)$ and $p : P$, suppose that $m < n \wedge mp_{\mathit{TSO}}(p, m, n, h)$. In the case where $h(g(n))$ is the return of a non-flush operation, $mp(p, g(m), g(n), h)$ holds so we know $f_1(g(n)) = f_1(g(m)) + 1$ (property of *lin*). G does not change this relationship between $f_1(g(n))$ and $f_1(g(m))$. Hence, $f(n) = f(m) + 1$.
On the other hand if $h(g(n))$ is the return of a flush operation, G moves $f_1(g(m))$ and $f_1(g(m+1))$ to $f_1(g(n-1))$ and $f_1(g(n))$ respectively. Again, we get $f(n) = f(m) + 1$.

(iii) Given $m, n, m', n' : 1 \,..\#\mathit{Trans}(h)$ and $p, q : P$ such that $n < m' \wedge mp_{\mathit{TSO}}(p, m, n, h) \wedge mp_{\mathit{TSO}}(q, m', n', h)$, it follows that $mp(p, g(m), g(n), h) \wedge mp(q, g(m'), g(n'), h)$. This means $f_1(g(n)) < f_1(g(m'))$ (property of *lin*). G does not change this relationship between $f_1(g(n))$ and $f_1(g(m'))$. Hence, $f(n) < f(m')$.

7 Conclusions

In this paper we have developed a method by which to simplify proofs of linearizability for algorithms running on the TSO memory model. Instead of having to deal with the effects of both fine-grained atomicity and local buffers in one set of proof obligations, we have used an intermediate specification to partition the proof obligations in two. One set of proof obligations is simply the standard existing notion of linearizability (where flushes are treated as normal operations), and any existing proof method can be employed to verify this step (we in fact use our mechanised simulation-based method). The second set of proof obligations involves verifying that an appropriate transformation (given by *TRANS* defined in Section 5) holds.

Although there is existing work on defining linearizability on TSO, to the best of our knowledge this is the first work that provides simplified reasoning for showing *how* linearizability can be verified for algorithms running on TSO, although mention should be made of the approach in [19] that uses SPIN to check specific runs for TSO-linearizability. Clearly this work could be extended in a number of directions. Specifically, we would like to mechanise the proof obligations inherent in *TRANS* using KIV in the same way that the existing proof methods for standard linearizability, such as those in [5–8, 14, 15], have already been encoded in the theorem prover. Additionally, we aim to look at the issue of completeness and related to this will be how one can *calculate* the required intermediate description from the concrete algorithm and abstract and concrete state spaces.

References

1. J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F.Z. Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In L. Petersen and M.M.T. Chakravarty, editors, *DAMP '09*, pages 13–24. ACM, 2008.
2. D. Amit, N. Rinetzky, T.W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In W. Damm and H. Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.
3. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In H. Seidl, editor, *ESOP 2012*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.
4. C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In H.R. Nielson and G. Filé, editors, *SAS 2007*, volume 4634 of *LNCS*, pages 233–238. Springer, 2007.
5. J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In J. Davies and J. Gibbons, editors, *IFM 2007*, volume 4591 of *LNCS*, pages 195–214. Springer, 2007.
6. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanizing a correctness proof for a lock-free concurrent stack. In G. Barthe and F.S. de Boer, editors, *FMOODS 2008*, volume 5051 of *LNCS*, pages 78–95. Springer, 2008.
7. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.
8. J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisabilty with potential linearisation points. In M. Butler and W. Schulte, editors, *FM 2011*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.
9. J. Derrick, G. Smith, and B. Dongol. Verifying linearizability on TSO architectures. In *iFM 2014*, volume 8739 of *LNCS*, pages 341–356, 2014.
10. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In D. de Frutos-Escrig and M. Nunez, editors, *FORTE 2004*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.
11. A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In M. Aguilera, editor, *DISC 2012*, volume 7611 of *LNCS*, pages 31–45. Springer, 2012.
12. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
13. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In *Automated Deduction*, pages 13–39. Kluwer, 1998.
14. G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In *CAV'12*, volume 7358 of *LNCS*, pages 243–259, 2012.
15. G. Schellhorn, H. Wehrheim, and J. Derrick. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. on Computational Logic*, 2014.
16. P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli, and M.O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.
17. D.J. Sorin, M.D. Hill, and D.A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.
18. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.
19. O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC2013*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.
20. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.