Reasoning Algebraically about Refinement on TSO Architectures

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

[1] Department of Computing, University of Sheffield, UK
[2] School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

Abstract. The Total Store Order memory model is widely implemented by modern multicore architectures such as x86, where local buffers are used for optimisation, allowing limited forms of instruction reordering. The presence of buffers and hardware-controlled buffer flushes increases the level of non-determinism from the level specified by a program, complicating the already difficult task of concurrent programming. This paper presents a new notion of refinement for weak memory models, based on the observation that pending writes to a process' local variables may be treated as if the effect of the update has already occurred in shared memory. We develop an interval-based model with algebraic rules for various programming constructs. In this framework, several decomposition rules for our new notion of refinement are developed. We apply our approach to verify the spinlock algorithm from the literature.

1 Introduction

Logics for reasoning about concurrency in shared memory systems are based on the assumption that hardware is *sequentially consistent* [18], guaranteeing that instructions within each process are never executed out of order in memory. However, modern processors have abandoned sequential consistency in favour of weaker memory guarantees, using local buffers to offer greater scope for optimisation. There are several different weak memory models [1, 2, 23]; in this paper, we focus on the most restricted of these: the *Total Store Order* (TSO) memory model, which is implemented by architectures such as x86 (see Fig. 1). Under TSO, instead of committing writes immediately to main memory, the process executing the write stores it as a pending write in its local buffer. Pending writes are not visible to other processors until they are *flushed*, which commits the write to shared memory. A flush is either programmer controlled (via instructions such as `fence` or `lock`) or hardware controlled. Programmer-controlled flushes are ultimately expensive (and inefficient), hence, one would like to keep these to a minimum. On the other hand, reasoning about hardware-controlled flushes is difficult due to the increase in non-determinism of a program's behaviour.

Several approaches to program verification under TSO have been developed; we provide a brief survey. Researchers have considered direct methods, such as executable memory models [22], theorems for reduction [9], and identification of race conditions [20]. Others have linked programs under TSO executions to an abstract specification using linearizability [7, 16], however, these use abstract specifications different from the

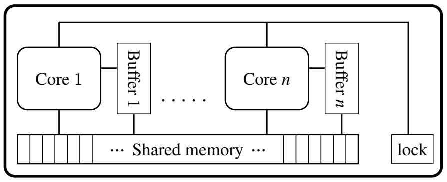

Fig. 1. TSO hardware overview

natural abstractions one would expect; [7] requires buffers to be present in the abstract specification, while [16] uses a non-deterministic abstract specification.

An issue with many existing approaches is that program semantics is given at a low level of abstraction of individual read and writes, which means programs must be understood and analysed using a verbose representation. Our work is based on the desire to lift reasoning to higher levels of abstraction [15], which in turn improves scalability. To this end, we develop an interval-based semantics by adapting Interval Temporal Logic [19]. Such an approach has two distinct advantages: (a) it allows one to define *truly concurrent* executions [10, 11], providing a more accurate model of TSO-based hardware; and (b) it is amenable to algebraic reasoning [3, 13], which enables one to develop *algebraic laws* for syntactically manipulating formulas representing program behaviour. In this paper, we develop algebraic rules to verify *refinement* between a concrete program and its abstract representation. The development of algebraic laws is non-trivial. However, once available, they provide high-level reusable theories for verification. We do not claim to have a complete set of laws (this is a topic of future work), instead, we provide a set of rules that are required for proving the spinlock example we verify.

Within this interval-based logic, we develop a framework for reasoning on TSO, simplifying our existing semantics [15] and introducing enhancements specifically designed for reasoning about buffer-based programs. This includes a simplified permission framework (Section 3.1), a novel methodology for evaluating expressions in the presence of local buffers (Section 4.3) and a novel notion of *local buffer refinement* (Section 5.2). Local buffer refinement is based on the observation that: To show a command C refines another command A with respect to a process p, the pending writes to local variables of p may be treated as if they have already taken effect in A. Thus, local updates at the concrete level may be treated as if they occur in their program order (without waiting for their flush to occur). This benefits verification because the non-determinism from flushes of local variables is resolved earlier. We develop a number of algebraic transformation laws for both refinement and local buffer refinement.

2 Background

2.1 Total Store Order Example

Total Store Order (TSO) memory allows a process to store a write in its local buffer and continue processing without waiting for this write to be commited to memory (i.e., while the write is pending). The values in the buffer are flushed in a FIFO order. To see

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

Fig. 2. Spinlock algorithm

```
Noncritical section ;
acquire();
Critical section ;
release();
```

Fig. 3. Spinlock client (a)

```
Noncritical section;
if tryacquire() {
   Critical section ;
   release() ;
}
```

Fig. 4. Spinlock client (b)

the effect of this, consider the following classic example with processes p and q that modify shared variables x and y, which are initialised to 0. In this paper, we assume maximum parallelism and that each thread resides in exactly one core, therefore, the words *process* and *core* are used synonymously.

```
word x=0, y=0;
p {  p1:   x := 1 ;                q {  q1:   y := 1 ;
     p2:   r1 := y }                    q2:   r2 := x }
```

Under sequentially consistent memory, at the end of execution, at least one of r1 or r2 would have a value 1. However, in TSO memory, it is possible to end execution so that both r1 and r2 read the original values of x and y, i.e., both r1 and r2 are 0 at termination. One such execution is $\langle \mathtt{p1}, \mathtt{p2}, \mathtt{q1}, \mathtt{q2}, \mathtt{flush}(\mathtt{p}), \mathtt{flush}(\mathtt{p}), \mathtt{flush}(\mathtt{q}), \mathtt{flush}(\mathtt{q})\rangle$, where $\mathtt{flush}(\mathtt{p})$ denotes a (hardware-controlled) flush event for process *p*. The write to x at p1 is not seen by process q until p's buffer is flushed, and symmetrically for the write to y at q1. Hence, it is possible for q to read a value 0 for x at q2 even though q2 is executed after p1.

In addition to the above behaviour, each TSO process reads pending writes from its own buffer if possible, and hence, may obtain values that are not yet globally visible to other processes, e.g., if p2 is replaced with r1 := x, process p would read x = 1 even if the write to x is pending. If there are multiple pending writes to the same location, then the write value corresponding to the last pending write is returned.

2.2 Case Study: Spinlock

Spinlock [4] is a locking mechanism designed to avoid operating system overhead associated with process scheduling and context switching. A typical implementation of spinlock is shown in Fig. 2, where a global variable x represents the lock and is set to 0 when the lock is held by a process, and 1 otherwise. The lock x is itself acquired using a secondary hardware lock (see Fig. 1), and this hardware lock is acquired/released using lock/unlock instructions. A process trying to acquire the lock x *spins*, i.e., waits in a loop and repeatedly checks the lock for availability.

Operation `acquire` only terminates if it successfully obtains the lock *x*. It will first lock the hardware so that no other process can access `x`. If, another process has already acquired *x* (i.e., `x = 0`) then it will release the hardware lock at `a7` and spin at `a8`, i.e., loop in the while-loop until *x* becomes free, before starting over from `a2`. Otherwise, it acquires the lock at `a4` by setting `x` to `0`, releases the hardware lock at `a5` and returns at `a6`. The operation `release` releases the lock by setting `x` to 1. The operation `tryacquire` is similar to `acquire`, but unlike `acquire` it only makes one attempt to acquire the lock. If this is successful it returns `true`, otherwise it returns `false`. Under TSO, a process *p* executing an assignment (e.g., `x := 0`) places a pending write in *p*'s local buffer, which is not visible to other processes until the buffer is flushed.

We refer to processes that use spinlock to provide mutual exclusion to a critical section of code as its *clients*. Here, as in [22], we assume that clients of the spinlock behave either as the program in Fig. 3 or Fig. 4. Thus, one can assume that a client only calls a `release` operation when it holds the lock.[3] Note however, that the behaviours in Fig. 3 and Fig. 4 are not exhaustive. To admit other behaviours, one may formalise the additional client code, then apply our proof methods in this paper to verify this additional behaviour.

Clients can ensure mutual exclusion in the critical sections if in place of `acquire`, `release` and `tryacquire`, they use *abstract operations* `AAcq`, `ARel` and `ATry` below, respectively, which do not use buffers. We will refer to such clients as *abstract clients*.

```
word x = 1;

void AAcq() {           void ARel() {           bool ATry() {
  await (x = 1) {         x <== 1                 if CAS(x, 1, 0) {
      x <== 0           }                           return true
  } }                                             else return false } }
```

Here, statement `await` denotes a blocking atomic test-and-set statement, e.g., `AAcq()` can only execute if `x = 1` holds, and its execution atomically sets the value of `x` to `0`. Unlike the concrete program in Fig. 2, all reads and writes occur directly with main memory; we use assignments of the form `x <== 0` (which directly updates the value of *x* in memory) to distinguish this difference. If `x = 0`, then `AAcq()` blocks and cannot execute further until its guard `x = 1` is set to true by another process. Operation `ATry()` attempts to update `x` to `0` using a non-blocking atomic compare-and-swap operation `CAS`, and returns 1 if the operation is successful and `0`, otherwise.[4]

Our notion of correctness of the spinlock will be to show that every possible execution of a spinlock client is a possible execution of an abstract client. To this end, we prove *refinement* between the behaviour of the two executions (see Section 5.3). Proving refinement under TSO is difficult; one must not only verify concurrency effects, but additionally consider the effect of accessing the buffer during a program's execution. Furthermore, the level of atomicity at which these effects are visible is fine-grained,

[3] Such restrictions on client behaviour must be made due to the simplicity of the spinlock algorithm in Fig. 2. Arbitrary client behaviour e.g., two consecutive calls to `release` without acquiring the lock will result in incorrect behaviour under TSO.

[4] `CAS(a, b, c)` is equivalent to `atomic { if a = b then a := c ; return true else return false}`.

occuring at the level of individual reads and writes. This paper develops a high-level approach for proving refinement that avoids the need to consider low-level (fine-grained) effects whenever possible by developing an interval-based semantics for programs under TSO. This allows one to view the concurrent execution of two processes as the conjunction of their behaviours over an interval (as opposed to an interleaving of their traces), reducing the impact of non-determinism due to concurrency.

3 Interval-Based Reasoning

3.1 Permission Monitoring

Using an interleaved execution semantics, one can guarantee that a variable will not be simultaneously written, or read and written as part of the same transition. This is not true for shared-memory true concurrency, where one must model variable access by the different processes (e.g., two processes simultaneousy modifying variable `x` in Fig. 2).

Our solution is to explicitly define read/write permissions. To this end, we assume that programs are executed by processes from a set *Proc*; each process represents a concurrent thread which modifies a set of variables from a set *Var*. The TSO architecture uses sophisticated coherence protocols to provide an illusion of shared memory. One may assume the following about read and write instructions:

- Two simultaneous writes (by different processes) to the same variable do not occur.
- A simultaneous read and write of the same variable does not occur.
- A process never has access permission to the local variable of another process.

As we shall see in Sections 4.2 and 4.4, permissions also provide a convenient mechanism for formalising the effect of a `lock-unlock` block.

In previous work [11], we have modelled permissions using a fractional encoding (inspired by [5]). Here, we simplify these general notions and define the *permission space* as $Perm \mathrel{\widehat{=}} Proc \to Var \to \mathbb{P}\{\mathsf{wr}, \mathsf{rd}\}$, where wr and rd denote write and read permission, respectively. Using '.' for function application, given $\pi \in Perm$, we interpret $\mathsf{wr} \in \pi.p.v$ (resp., $\mathsf{rd} \in \pi.p.v$) as $p \in Proc$ has permission to write to (resp., read the value of) $v \in Var$.

A system at any time is described by a state of type $State \mathrel{\widehat{=}} Var \to Val$, where *Val* is the set of values. The system over time is formalised by a *stream*, which is a total function of type $Stream \mathrel{\widehat{=}} \mathbb{Z} \to (State \times Perm)$. Therefore, for each time in $\mathbb{Z}$, a stream formalises the state of the system and the permissions for each process and variable. Properties of a system are given by *predicates*; a predicate of type T is a member of $\mathcal{P}T \mathrel{\widehat{=}} T \to \mathbb{B}$, e.g., $\mathcal{P}State$, $\mathcal{P}Stream$, and $\mathcal{P}Perm$ are state, stream, and permission predicates, respectively. We assume pointwise lifting of boolean operators over predicates in the normal manner.

If $\pi \in Perm$, then $\mathcal{W}.p.v.\pi \mathrel{\widehat{=}} (\mathsf{wr} \in \pi.p.v)$, $\mathcal{R}.p.v.\pi \mathrel{\widehat{=}} (\mathsf{rd} \in \pi.p.v)$ and $\mathcal{N}.p.v.\pi \mathrel{\widehat{=}} (\pi.p.v = \varnothing)$ denote permission predicates that hold iff process p has write, read or no access to v in the permission space π, respectively.

Example 1. Suppose $Var = \{u, v\}$, $Proc = \{p, q\}$ and

$$\pi \mathrel{\widehat{=}} \{p \mapsto \{u \mapsto \{\mathsf{rd}, \mathsf{wr}\}, v \mapsto \{\mathsf{rd}\}\}, q \mapsto \{u \mapsto \varnothing, v \mapsto \{\mathsf{rd}\}\}\}$$

Then, $\mathcal{W}.p.u.\pi$ (p has write permission to u), $(\mathcal{R}.p \wedge \mathcal{R}.q).v.\pi$ (both p and q have read permission to v) and $\mathcal{N}.q.u.\pi$ (q has no permission to u) in space π. Note that due to pointwise lifting $(\mathcal{R}.p \wedge \mathcal{R}.q).v.\pi = \mathcal{R}.p.v.\pi \wedge \mathcal{R}.q.v.\pi$. $\square$

The assumptions on reads and writes above are then taken into account by assuming that each *valid* permission space π satisfies the following, where p and q such that $p \neq q$ are processes, v is a variable, and u_p is a local variable of process p.

$$(\mathcal{W}.p.v.\pi \Rightarrow \mathcal{N}.q.v.\pi) \wedge (\mathcal{R}.p.v.\pi \Rightarrow \neg\mathcal{W}.q.v.\pi) \wedge \pi.p.u_p = \{\mathsf{wr}, \mathsf{rd}\} \qquad (1)$$

Note that the third conjunct combined with the first ensures that q does not have read nor write permission to any local variable of p. This is lifted to the level of streams by defining a *valid* stream to be one in which each $s.t$ is valid for $t \in \mathbb{Z}$. For the rest of the paper, we assume each stream is valid.

To simplify the notation, for a state predicate b and permission predicate z, we assume $b.(\sigma, \pi) = b.\sigma$ and $z.(\sigma, \pi) = z.\pi$, where σ and π are a state and a permission state, respectively. We assume '$\upharpoonright$' denotes a projection operator, e.g., $(x, y) \upharpoonright 1 = x$.

3.2 Interval Predicates

In this section, we provide the basics of interval predicates, which forms the logical basis of our program semantics. Our logic is an adaptation of Interval Temporal Logic [19]. An *interval* is a contiguous set of integers (denoted $\mathbb{Z}$), and hence the set of all intervals is $\mathit{Intv} \mathrel{\widehat{=}} \{\Delta \subseteq \mathbb{Z} \mid \forall t_1, t_2 \colon \Delta \bullet \forall t \colon \mathbb{Z} \bullet t_1 \leq t \leq t_2 \Rightarrow t \in \Delta\}$.

We let $\mathsf{lub}.\Delta$ and $\mathsf{glb}.\Delta$ denote the *least upper* and *greatest lower* bounds of an interval Δ, respectively. Furthermore, we define $\mathsf{inf}.\Delta \mathrel{\widehat{=}} (\mathsf{lub}.\Delta = \infty)$, $\mathsf{fin}.\Delta \mathrel{\widehat{=}} \neg\mathsf{inf}.\Delta$, and $\varepsilon.\Delta \mathrel{\widehat{=}} (\Delta = \varnothing)$. We define an ordering $\Delta_1 < \Delta_2 \mathrel{\widehat{=}} \forall t_1 \colon \Delta_1, t_2 \colon \Delta_2 \bullet t_1 < t_2$. To facilitate reasoning about specific parts of a stream, we use *interval predicates*, which have type $\mathit{IntvPred} \mathrel{\widehat{=}} \mathit{Intv} \to \mathcal{P}\mathit{Stream}$ [11, 13].

Example 2. Given *Var*, *Proc* and π as defined in Example 1, we define

$$\begin{array}{rlrl}
\sigma_1 & \mathrel{\widehat{=}} \{u \mapsto 500, v \mapsto 42\} & s & \mathrel{\widehat{=}} \lambda\, t \bullet \ \mathsf{if}\, t \geq 10\ \mathsf{then}\ (\sigma_1, \pi)\ \mathsf{else}\ (\sigma_2, \pi) \\
\sigma_2 & \mathrel{\widehat{=}} \{u \mapsto 0, v \mapsto 1\} & g & \mathrel{\widehat{=}} \lambda\, \Delta \bullet\ \lambda\, s \bullet \forall t \colon \Delta \bullet ((s.t) \upharpoonright 1).u \geq 300 \\
b & \mathrel{\widehat{=}} \lambda\, \sigma \bullet \sigma.u < \sigma.v & &
\end{array}$$

Then, σ_1, σ_2 are states, b a state predicate, s is a stream and g is an interval predicate. Each of $(\neg b).(\sigma_1, \pi)$, $b.(\sigma_2, \pi)$, $\neg g.[-3, 3].s$ and $g.[10, 100).s$ hold.[5] $\square$

We define *universal implication* $g_1 \Rrightarrow g_2 \mathrel{\widehat{=}} \forall \Delta \colon \mathit{Intv}, s \colon \mathit{Stream} \bullet g_1.\Delta.s \Rightarrow g_2.\Delta.s$ for interval predicates g_1 and g_2, and write $g_1 \equiv g_2$ iff both $g_1 \Rrightarrow g_2$ and $g_2 \Rrightarrow g_1$ hold.

4 Concurrent Programming with Intervals

4.1 Operators to Model Programming Constructs

In this section, we introduce interval predicate operators used to formalise common programming constructs: sequential composition, branching, loops, and parallel composition. To model sequential composition, we define the *chop* operator [19, 13]. Unlike

[5] Here, $[-3, 3]$ is the closed interval from -3 to 3 and $[10, 100)$ is the right-open interval from 10 to 100.

Interval Temporal Logic, which requires adjoining intervals to overlap at a single point, adjoining intervals in our logic are disjoint.

$$\begin{aligned}(g_1 \mathbin{;} g_2).\mathit{\Delta}.s \mathrel{\widehat{=}} & \big(\exists \mathit{\Delta}_1, \mathit{\Delta}_2 \bullet (\mathit{\Delta}_1 \cup \mathit{\Delta}_2 = \mathit{\Delta}) \wedge \mathit{\Delta}_1 < \mathit{\Delta}_2 \wedge g_1.\mathit{\Delta}_1.s \wedge g_2.\mathit{\Delta}_2.s\big) \vee \\ & (\mathsf{inf} \wedge g_1).\mathit{\Delta}.s\end{aligned}$$

Thus, $(g_1 \mathbin{;} g_2).\mathit{\Delta}.s$ holds iff either interval $\mathit{\Delta}$ may be split into two adjoining parts $\mathit{\Delta}_1$ and $\mathit{\Delta}_2$ so that g_1 holds for $\mathit{\Delta}_1$ and g_2 holds for $\mathit{\Delta}_2$ in s, or the least upper bound of $\mathit{\Delta}$ is ∞ and g_1 holds for $\mathit{\Delta}$ in s. Inclusion of the second disjunct $(\mathsf{inf} \wedge g_1).\mathit{\Delta}.s$ enables g_1 to model an infinite (divergent or non-terminating) program. We assume that ';' binds tighter than all other binary operators, e.g., $g_1 \mathbin{;} g_2 \vee h = (g_1 \mathbin{;} g_2) \vee h$.

Example 3. For b, g and s as defined in Example 2, if $h \mathrel{\widehat{=}} \lambda\,\mathit{\Delta} \bullet \lambda\, s \bullet \exists t \colon \mathit{\Delta} \bullet b.(s.t)$, then $(h \mathbin{;} g).[0, 100).s$ holds because both $h.[0, 10).s$ and $g.[10, 100).s$ hold. Note that there may be more than one possible way to split up an interval when applying the definition of chop. □

Non-deterministic choice is modelled by (lifted) disjunction, and hence, for example, the behaviour of `if b then S1 else S2` is given by *test*.`b` ; *beh*.`S1` $\vee$ *test*.($\neg$`b`) ; *beh*.`S2`, where *test*.`b` and *beh*.`S1` are interval predicate formalisations of evaluating `b` and executing `S1`, respectively. The precise value of *test*.`b` depends on the atomicity assumptions of the program under consideration, and hence, the interpretation of *test*.`b` is non-trivial (see [17, 13, 11]). The value of *test*.`b` is modelled by command $[b]$ (see Section 4.2), whereas at the concrete level its value is formalised by a different command $[\![b]\!]$, which takes the effect of the buffer into account (see Section 4.3).

Iteration g^* and g^ω are the least and greatest fixed points of $\lambda\, z \bullet gz \vee \varepsilon$, respectively [14], where g^* allows empty and finite iterations and g^ω allows empty, finite and infinite iterations of g. We also define strictly finite and possibly infinite *positive* iterations.

$$\begin{aligned} g^* &\mathrel{\widehat{=}} \mu z \bullet ((g \mathbin{;} z) \vee \varepsilon) & \qquad g^+ &\mathrel{\widehat{=}} g \mathbin{;} g^* \\ g^\omega &\mathrel{\widehat{=}} \nu z \bullet ((g \mathbin{;} z) \vee \varepsilon) & \qquad g^{\omega+} &\mathrel{\widehat{=}} g \mathbin{;} g^\omega \end{aligned}$$

A thorough algebraic treatment of loops using iteration is given in [3]. For example, program code `while b do S` is modelled by $(\mathit{test}.b \mathbin{;} \mathit{beh}.S)^\omega \mathbin{;} \mathit{test}.(\neg b)$. In this paper, we use the following rule.

Law 1 (Leapfrog [3]) *For interval predicates g and h, both* $g \mathbin{;} (h \mathbin{;} g)^\omega \equiv (g \mathbin{;} h)^\omega \mathbin{;} g$ *and* $g \mathbin{;} (h \mathbin{;} g)^* \equiv (g \mathbin{;} h)^* \mathbin{;} g$ *hold.*

We are interested in modelling true concurrency and therefore simply treat the parallel composition of two or more processes using (lifted) logical conjunction. For example, the behaviour of $g_1 \mathbin{;} g_2$ in parallel with $h_1 \mathbin{;} h_2$ over an interval $\mathit{\Delta}$ in stream s is given by $(g_1 \mathbin{;} g_2 \wedge h_1 \mathbin{;} h_2).\mathit{\Delta}.s$. Using pointwise lifting, this is equivalent to $(g_1 \mathbin{;} g_2).\mathit{\Delta}.s \wedge (h_1 \mathbin{;} h_2).\mathit{\Delta}.s$, which holds iff (a) $\mathit{\Delta}$ can be split into adjoining intervals $\mathit{\Delta}_1$ and $\mathit{\Delta}_2$ such that $g_1.\mathit{\Delta}_1.s \wedge g_2.\mathit{\Delta}_2.s$ holds; and (b) $\mathit{\Delta}$ can also be split into adjoining intervals $\mathit{\Delta}_3$ and $\mathit{\Delta}_4$ such that $h_1.\mathit{\Delta}_3.s \wedge h_2.\mathit{\Delta}_4.s$. Note that there is no immediate correlation between the lengths of $\mathit{\Delta}_1$ and $\mathit{\Delta}_3$, i.e. g_1 could terminate earlier than h_1, and vice versa.

Modelling tests. Interval predicates provide a flexible approach to non-deterministic state predicate evaluation [17], where expression evaluation is assumed to take time (as opposed to being instantaneous). In this paper, guards and assignments are restricted to contain at most one shared variable.[6] Given that c is either a state or permission predicate, and $\varDelta$ and s are an interval and stream, respectively, we define:

$$(\boxdot\, c).\varDelta.s \mathrel{\widehat=} \neg\varepsilon.\varDelta \wedge \forall t \colon \varDelta \bullet c.(s.t)$$

Thus, $(\boxdot\, c).\varDelta.s$ holds iff $\varDelta$ is non-empty and c holds for each state of s within $\varDelta$. For example, $\neg\varepsilon \wedge g \equiv \boxdot(u \geq 300)$, where g is the interval predicate defined in Example 2.

Reasoning about pre/post assertions. One may define several additional interval predicate operators [13]. For the purposes of this paper, we find it useful to reason about properties that hold in the immediately preceding interval. We therefore define

$$(\ominus\, g).\varDelta.s \mathrel{\widehat=} \neg\varepsilon.\varDelta \wedge \mathsf{glb}.\varDelta \neq -\infty \wedge g.(\mathrm{prev}.\Delta).s$$

where $\mathrm{prev}.\varDelta \mathrel{\widehat=} \{t \colon \mathbb{Z} \mid \forall u \colon \varDelta \bullet t < u\}$ is the interval of all times before $\varDelta$. If c is a state or permission predicate, we use notation $\overrightarrow{c} \mathrel{\widehat=} \mathit{true}\,;\boxdot\, c$, where $\overrightarrow{c}.\varDelta$ states that c holds at the end of $\varDelta$ whenever $\mathsf{inf}.\varDelta \neq \infty$. Additionally, we define the following notation to reason about assertions that immediately precede, or are a result of a computation.

$$\{c\}g \mathrel{\widehat=} \ominus\overrightarrow{c} \wedge g \qquad\qquad g\{c\} \mathrel{\widehat=} g \wedge \overrightarrow{c}$$

Such a definition of a pre-assertion is necessary because we assume adjoining intervals do not overlap (unlike [19]). We have the following useful properties, which can be proved in a straightforward manner.

$$\{c\}(g_1 \vee g_2) \equiv (\{c\}g_1) \vee (\{c\}g_2) \tag{2}$$

$$g_1\{c\}\,;\, g_2 \equiv g_1\,;\,\{c\}g_2 \qquad \text{provided } g_1 \vee g_2 \Rrightarrow \neg\varepsilon \tag{3}$$

4.2 Abstract Commands

Using the interval-based semantic basis from the previous sections, we formalise *commands*, which describe the behaviours of the system processes. Formally, a command is of type $\mathit{Cmd} \mathrel{\widehat=} \mathbb{P}_1\, \mathit{Proc} \to \mathit{IntvPred}$, mapping non-empty sets of processes to an interval predicate representing their behaviour. We use $C.p$ as shorthand for $C.\{p\}$, where C is a command and p is a process.

The semantics of sequential composition, iteration, non-deterministic choice and parallel composition of commands are defined pointwise lifting of the interval predicate operators, and hence, are given in the same syntax, e.g., $(C_1\,;C_2).p = C_1.p; C_2.p$. What remains is to define the commands to model, say, guard evaluation and assignment.

We first present some basic commands that may be used to models the abstract (sequentially consistent) specification. In particular, we define *idling* (denoted id), *abstract guard evaluation* (denoted $[b]$), *memory update* (denoted $\overline{v} \Lleftarrow \overline{e}$) and *locked access* (denoted $\boxed{\overline{v} \bullet C}$), where $\overline{v}$ and $\overline{e}$ denote vectors of variables and expressions, respectively. We define $\mathit{Deny}.\overline{v}.p \mathrel{\widehat=} \boxdot(\forall q \colon \mathit{Proc}\backslash\{p\}, v \colon \overline{v} \bullet (\neg\mathcal{W} \wedge \neg\mathcal{R}).q.v)$, which states that the variables in $\overline{v}$ are not accessed by processes other than p. We assume that $\mathit{vars}.b$ denotes the free variables in b.

$$\begin{aligned} \mathsf{nid}.p &\mathrel{\widehat=} \forall v \colon \mathit{Var} \bullet \boxdot\, \neg\mathcal{W}.p.v \\ \mathsf{id}.p &\mathrel{\widehat=} \neg\varepsilon \Rightarrow \mathsf{nid}.p \\ [b].p &\mathrel{\widehat=} \boxdot b \wedge \mathsf{nid}.p \wedge \\ &\qquad \forall v \colon \mathit{vars}.b \bullet \boxdot\, \mathcal{R}.p.v \end{aligned} \qquad \begin{aligned} (\overline{v} \Lleftarrow \overline{e}).p &\mathrel{\widehat=} \exists \overline{k} \bullet \{\overline{e} = \overline{k}\} \\ &\qquad \boxdot(\overline{v} = \overline{k}) \wedge \boxdot(\forall v \colon \overline{v} \bullet \mathcal{W}.p.v) \\ \boxed{\overline{v} \bullet C}\,.p &\mathrel{\widehat=} (\overline{v} \neq \varnothing \Rightarrow \mathit{Deny}.\overline{v}.p) \wedge C.p \wedge \\ &\qquad (\forall v \colon \overline{v} \bullet \boxdot(\mathcal{W} \wedge \mathcal{R}).p.v) \end{aligned}$$

[6] This can be extended to handle more complex expressions [17].

Thus $\mathsf{nid}.p$ states that p is *write idle* i.e., p does not have write access to any variable during the given (non-empty) interval; $\mathsf{id}.p$ states that either p is write idle or the interval under consideration is empty; $[b].p$ holds iff b holds throughout the given interval and p is idle; $\overline{v} \Leftarrow \overline{e}$ denotes an instantaneous update, which holds iff $\overline{e}$ evaluates to a vector of values $\overline{k}$ as a pre-assertion, and $\overline{v}$ is updated to $\overline{k}$, where p has write permission to each $v \in \overline{v}$; and $\boxed{\overline{v} \bullet C}.p$ holds iff $C.p$ holds and additionally no process other than p has permission to access $v \in \overline{v}$.

Example 4. The abstract specification is formalised as follows, where *AAcq* and *ARel* specify operations `AAcq` and `ARel`, respectively, while *ATryOK* and *ATryFl* specify execution of the `ATry` operation that succeed and fail to acquire the lock, respectively. We abbreviate `x = 1` and `x = 0` to x and $\neg x$, respectively. The return value of an execution of `tryacquire` in process p is modelled by a local variable r_p.

$$AAcq.p \mathrel{\widehat{=}} \boxed{x \bullet [x] ; (x \Leftarrow 0)} \qquad ATryOK.p \mathrel{\widehat{=}} AAcq.p ; \mathsf{id} ; (r_p \Leftarrow true)$$
$$ARel.p \mathrel{\widehat{=}} x \Leftarrow 1 \qquad ATryFl.p \mathrel{\widehat{=}} \boxed{x \bullet [\neg x]} ; \mathsf{id} ; (r_p \Leftarrow false)$$
$$AExec.p \mathrel{\widehat{=}} ((AAcq ; \mathsf{id} ; ARel) \vee (ATryOk ; \mathsf{id} ; ARel) \vee ATryFl).p$$
$$Spec.P \mathrel{\widehat{=}} \{x\} \textstyle\bigwedge_{p:P}((\mathsf{id} ; AExec)^{\omega} ; \mathsf{id}).p$$

The concurrent execution of abstract clients is modelled by *Spec*, which begins in a state in which the lock `x` is available (i.e., x holds) and consists of a number of (truly) parallel processes. We assume that each client of the spinlock behaves as either Fig. 3 or Fig. 4, and furthermore, that the critical and non-critical sections do not modify variables x and r_p, and hence, both the critical and non-critical sections are modelled by id. Therefore, $\mathsf{id} ; AExec$ models a single call to the abstract spinlock. Each process may make multiple (zero or more) calls, followed by no calls, and hence, all possible behaviours of an abstract client is given by $(\mathsf{id} ; AExec)^{\omega} ; \mathsf{id}$.

We now explain how each operation is modelled. If $AAcq.p.\Delta.s$ holds for interval Δ and stream s, then only process p has access to x (i.e., no process $q \neq p$ may read or write to x) and either (i) Δ can be partitioned into Δ_1 and Δ_2 with $\Delta_1 < \Delta_2$ such that x holds in s throughout Δ_1 and x is updated to 0 in s within Δ_2, or (ii) Δ is infinite and x (i.e., `x = 1`) holds in s throughout Δ. Because `await b` *blocks* until *test*.`b` becomes true, there are no behaviours for *AAcq.p* when *test*.(¬`b`) holds. Operation *ARel.p* immediately sets x to 1, and by the definition of $\Leftarrow$ together with assumption (1), we have that no other process reads or writes to x while this update occurs. Operation *ATryOK.p* behaves as *AAcq.p*, performs some idling, then updates r_p to *true*. The idling between *AAcq.p* and update to r_p provides scope for potential stuttering at the concrete level. Operation *ATryFl.p* starts by behaving as $\boxed{x \bullet [\neg x]}$, which implies that x is not accessed by any process $q \neq p$ and that $\neg x$ holds throughout the given interval. Then, *ATryFl.p* performs some idling and updates the return value r_p to *false*. □

4.3 Reading Variables for Expression Evaluation with Buffer Effects

Section 4.2 provided an interval-based semantics for commands without buffers, which were in turn used to model the abstract specification. The concrete program executes under TSO memory and contains local buffers, whose effects on the program's behaviour must be formalised. In this section, we present a method for evaluating expressions, i.e., when processes read variables, in the presence of local buffers. In particular, we formalise the fact that a TSO process first checks its buffer for pending writes; if a pending write exists, the last pending value is returned, and otherwise the value from memory is returned. Using interval-based methods enables one to formalise the effects of a buffer on the value of an expression at a high level of abstraction [15].

We assume that $\mathrm{B}_p \in \mathit{Var}$ denotes the buffer for process p, whose value is of type $\mathrm{seq}.(\mathit{Var} \times \mathit{Val})$, representing a pending write. Each buffer may contain multiple pending writes to the same location, and hence, we define a function *cover* that returns a set of mappings to the last pending write in a given buffer. Because $\mathrm{seq}.X$ is a partial function of type $\mathbb{N} \mathrel{+\!\!\!\!\rightarrow} X$, we may use $\mathrm{dom}.z$ to refer to the indices of $z \in \mathrm{seq}.X$.

$$\mathit{cover}.B \mathrel{\widehat{=}} \{B.i \mid i{:}\, \mathrm{dom}.B \land \forall j{:}\, \mathrm{dom}.B \bullet j > i \Rightarrow (B.i \restriction 1) \neq (B.j \restriction 1)\}$$

When a process evaluates an expression, the values of pending writes in a process' buffer *mask* those in memory, which is modelled formally using functional override '$\oplus$' (see [24] for a formal definition).

Example 5. Suppose B and BB are buffers (BB is not shown below), p and q are processes, u, v, and w are variables, and σ is a state such that

$$B \mathrel{\widehat{=}} \langle (v, 11), (w, 33), (v, 44)\rangle \qquad\qquad \sigma \mathrel{\widehat{=}} \{(\mathrm{B}_p, B), (\mathrm{B}_q, BB), (u, 0), (v, 1), (w, 2)\}$$

Then we have $\mathit{cover}.B = \{(w, 33), (v, 44)\}$, i.e., for each variable in B its last corresponding value in B is picked. Hence, we have

$$\sigma \oplus \mathit{cover}.B = \{(\mathrm{B}_p, B), (\mathrm{B}_q, BB), (u, 0), (v, 44), (w, 33)\}$$

which replaces each mapping in σ by those in *cover*.B. □

We lift buffer effects to state predicates using $(\mathit{mask}.b.B).\sigma \mathrel{\widehat{=}} b.(\sigma \oplus \mathit{cover}.B)$, which states that b holds in a state σ covered by B. For the definitions in Example 5, both $(\mathit{mask}.(u = 0).B).\sigma$ and $(\mathit{mask}.(w < v).B).\sigma$ hold, but $(w < v).\sigma$ does not.

Processes evaluate state predicates (e.g., as part of a guard), however, in the presence of permissions and local buffers, evaluation is non-trivial. Firstly, one must ensure that a process p evaluating state predicate b is able to obtain read permission to each variable of b whenever the variable's value is fetched from memory. Note that this is only potentially problematic if the variable in question is shared (i.e., not a local variable of p) and not in p's buffer (p may can always access its local buffer). Secondly, the value of a variable v read by p must be the last value of v in p's buffer if it exists, and the value of v in memory, otherwise. Assuming $\mathit{vars}.B \mathrel{\widehat{=}} \{(B.i) \restriction 1 \mid i \in \mathrm{dom}.B\}$ is the set of all variables in $B \in \mathrm{seq}.(\mathit{Var} \times \mathit{Val})$, we define:

$$\boxdot_p\, b \mathrel{\widehat{=}} \boxdot(\mathit{mask}.b.\mathrm{B}_p \land \forall v{:}\, \mathit{vars}.b \backslash \mathit{vars}.\mathrm{B}_p \bullet \mathcal{R}.p.v)$$

Thus, $(\boxdot_p\, b).\mathit{\Delta}.s$ models the evaluation of state predicate b by the process p, by either reading variables from p's buffer (if possible) or from main memory. Here $(\boxdot_p\, b).\mathit{\Delta}.s$ holds iff (i) b masked by B_p holds in s throughout $\mathit{\Delta}$, and (ii) p has read permission to the free variables of b not in $\mathit{vars}.\mathrm{B}_p$ throughout $\mathit{\Delta}$ in s.[7] In Section 4.4, $(\boxdot_p\, b)$ is

[7] In general, if b contains multiple shared variables, $\boxdot_p\, b$ is not an accurate model of evaluation because the variables in b may be read at different instants [17, 10, 12]. However, here, we assume that each expression/guard of each program under consideration contains at most one shared variable, in which case $\boxdot_p\, b$ is accurate (see [17, 10, 12]).

used to define expression evaluation, which in turn is used to model guards and the right hand side of assignments.

4.4 Commands under TSO

As already mentioned, processes that execute under TSO write only to their local buffers. The effects of these writes are not seen by other processes until a buffer is *flushed*, which moves the pending write from a buffer to shared memory. TSO buffers operate in a FIFO order, and hence, we define the following commands, where Φ models a single flush, $\widetilde{\Phi}$ models a flush or a non-empty idle, and Φ_{all} models a complete buffer flush.

$$\begin{array}{ll} \Phi\,.p \mathrel{\widehat{=}} \exists k \bullet \{\mathrm{B}_p = k \wedge k \neq \langle\,\rangle\} & \widetilde{\Phi} \mathrel{\widehat{=}} \Phi \vee \mathsf{nid} \\ \qquad\qquad \mathrm{B}_p, (k.0 \upharpoonright 1) \Lleftarrow \mathit{tail}.k, (k.0 \upharpoonright 2) & \Phi_{all}\,.p \mathrel{\widehat{=}} \Phi^{+}.p\,\{\mathrm{B}_p = \langle\,\rangle\} \end{array}$$

Due to the fine-granularity of the concrete implementation, seemingly atomic statements become compound commands under TSO memory. Evaluation of a boolean expression b (e.g. a guard in an `if-then-else` block) is a compound statement that flushes or idles (zero or more times), evaluates b using the buffer-based evaluation semantics defined in Section 4.3, then flushes or idles again (zero or more times). A write of v with value k appends the pair (v, k) to the end of the local buffer. An assignment to a constant value k, potentially flushes or idles (zero or more times), appends the value to the buffer, then potentially flushes or idles (zero or more times). An assignment to a complex expression e, first evaluates the expression to a value k, then assigns k to v. Thus, we define:

$$[\![b]\!].p \mathrel{\widehat{=}} \widetilde{\Phi}^{*}.p \,;\, (\boxdot_p\, b \wedge \mathsf{nid}.p) \,;\, \widetilde{\Phi}^{*}.p$$

$$(v \mapsfrom k).p \mathrel{\widehat{=}} \mathrm{B}_p \Lleftarrow \mathrm{B}_p \frown \langle (v, k) \rangle$$

$$v := e \mathrel{\widehat{=}} \mathsf{if}\ e \in \mathit{Val}\ \mathsf{then}\ \widetilde{\Phi}^{*} \,;\, (v \mapsfrom e) \,;\, \widetilde{\Phi}^{*}\ \mathsf{else}\ \exists k \colon \mathit{Val} \bullet [\![e = k]\!] \,;\, v := k$$

There are several TSO instructions that force the entire buffer to be flushed. These additionally may lock certain variables from being accessed while the flush all is being executed. We therefore define commands $\mathit{pre}_{\Phi}.\overline{v}.C.p$ and $\mathit{post}_{\Phi}.\overline{v}.C.p$, where $\mathit{pre}_{\Phi}.\overline{v}.C.p$ flushes the entire buffer (locking $\overline{v}$) before C is executed (and similarly $\mathit{post}_{\Phi}.\overline{v}.C.p$ flushes all after C).

$$\mathit{pre}_{\Phi}.\overline{v}.C.p \mathrel{\widehat{=}} \{\mathrm{B}_p = \langle\,\rangle\}C.p \vee (\boxed{\overline{v} \bullet \Phi_{all}} \,;\, C).p$$

$$\mathit{post}_{\Phi}.\overline{v}.C.p \mathrel{\widehat{=}} C.p\{\mathrm{B}_p = \langle\,\rangle\} \vee (C \,;\, \boxed{\overline{v} \bullet \Phi_{all}}).p$$

Some TSO instructions do not lock the memory while the buffer is being flushed. These may be modelled using $\mathit{pre}_{\Phi}.\varnothing.C$ and $\mathit{post}_{\Phi}.\varnothing.C$, which we abbreviate to $\mathit{pre}_{\Phi}.C$ and $\mathit{post}_{\Phi}.C$, respectively. A `lock` (e.g., `a2` in Fig. 2) acquires the memory lock then flushes the entire buffer; an `unlock` (e.g., `a5` in Fig. 2) flushes the entire buffer then releases the memory lock. Therefore, executing a command C within a `lock-unlock` block is modelled by

$$\boxed{\overline{v} \bullet C}_{\Phi} \mathrel{\widehat{=}} \mathit{pre}_{\Phi}.\overline{v}.(\mathit{post}_{\Phi}.\overline{v}.\boxed{\overline{v} \bullet C})$$

which executes C and ensures the buffer is empty before and after executing C. In addition it ensures that no reads and writes to $\overline{v}$ by other processes occur while C is being executed. Note however, that if a process p executes $\boxed{\overline{v} \bullet C}_{\Phi}$ and a process $q \neq p$ has a pending write to $\overline{v}$ in its local buffer, then q may read this value of $\overline{v}$ even while p is executing $\boxed{\overline{v} \bullet C}_{\Phi}$.

Example 6. Our modelling notation is used to formalising the behaviour of the concrete implementation as follows, where $Lck.p \mathrel{\widehat{=}} \boxed{x \bullet \llbracket x \rrbracket \ ; \ (x := 0)}_{\Phi}$.

$$\begin{array}{rlrl} Acq.p \mathrel{\widehat{=}} & \left(\boxed{x \bullet \llbracket \neg x \rrbracket}_{\Phi} \ ; \ \llbracket \neg x \rrbracket^{\omega} \ ; \ \llbracket x \rrbracket\right)^{\omega} \ ; \ Lck.p & Rel.p \mathrel{\widehat{=}} & x := 1 \\ TryOK.p \mathrel{\widehat{=}} & Lck.p \ ; \ (r_p := true) & TryFl.p \mathrel{\widehat{=}} & \boxed{x \bullet \llbracket \neg x \rrbracket}_{\Phi} \ ; \ (r_p := false) \\ Exec.p \mathrel{\widehat{=}} & (Acq \ ; \ \mathsf{id} \ ; \ Rel \vee TryOK \ ; \ \mathsf{id} \ ; \ Rel \vee TryFl).p & & \\ Prog.P \mathrel{\widehat{=}} & \{x\} \bigwedge_{p:P} (\{\mathrm{B}_p = \langle\rangle\}(\widetilde{\Phi}^{+} \ ; \ Exec)^{\omega} \ ; \ post_{\Phi}.x.(\widetilde{\Phi}^{+})).p & & \end{array}$$

Within $Acq.p$, command $\boxed{x \bullet \llbracket \neg x \rrbracket}_{\Phi}$ models an execution consisting of the lock at `a2`, failed test at `a3`, unlock at `a7` (see Fig. 2). Command $\llbracket \neg x \rrbracket^{\omega} \ ; \ \llbracket x \rrbracket$ models the while loop at `a8`. Therefore, the outermost $^{\omega}$ iteration in $Acq.p$ models executions of the outermost loop of `acquire` that fail to acquire the lock. Command $Lck.p$ models the lock at `a2`, successful test at `a3`, assignment at `a4`, and unlock at `a5` followed by the return at `a6`. The other operations are similar. □

5 Refinement and Local Refinement for TSO

5.1 Interval-Based Refinement

In this section, we develop a theory for proving that a command C *refines* another command A, providing a formal link between the behaviours of C and A. Here, A is an abstraction and therefore admits more behaviours than C, or conversely, any behaviour of C must also be a behaviour of A. In an interval-based setting, we use the following definition of refinement [11]. In the context of our example, if refinement holds, then whenever a spinlock client is able to enter its critical section, it must also be possible for the abstract client to enter the critical section.

Definition 1. *If C and A are commands, then C* refines *A* with respect to *$P \subseteq Proc$, (denoted $C \sqsupseteq_P A$) iff for any interval Δ and stream s, $C.P \Rrightarrow A.P$. We say C* is equivalent to *A* with respect to *P (denoted $C \sqsupseteq\sqsubseteq_P A$) iff both $C \sqsupseteq_P A$ and $A \sqsupseteq_P C$.*

Refinement is defined in terms of implication, and hence, relation $\sqsupseteq_P$ is both reflexive and transitive. In this paper, we use $\sqsupseteq_P$ as a basis for transforming the abstract specification *Spec* and the concrete program *Prog* individually. We use a notion of local buffer refinement (Definition 2) to relate concrete behaviours (with buffers) to abstract behaviours (without buffers).

Example 7. We transform the TSO implementation *Prog* into a form that is more amenable to verification. In particular, a difficulty encountered when verifying *Prog* in Example 6 directly is that for each process, p, command $Exec.p$ is not guaranteed to end in a flush, and hence changes to x may not be globally visible until the start of the next iteration (which starts with a `lock` that performs a flush all). In particular, it is not immediately possible to match the behaviour of *Rel* with abstract *ARel* because *Rel* only places a pending write in the buffer, whereas *ARel* modifies the value of x in memory. Therefore, we aim to transform *Prog* to *Prog′* below (see Proposition 1), where the flush occurs at the end of execution. We use notation

$$\boxed{\overline{v} \bullet C}_{\Phi} \mathrel{\widehat{=}} post_{\Phi}.\overline{v}.\boxed{\overline{v} \bullet C}$$

Unlike $\boxed{\overline{v} \bullet C}_{\Phi}$, command $\ulcorner \overline{v} \bullet C \lrcorner_{\Phi}$ only flushes the buffer at the end of execution. Below, we have used the property $g^{\omega} \equiv \varepsilon \vee g^{\omega+}$ to split *Acq* into two cases. Note that Acq'_1 is defined in terms of *Acq*.

$$\begin{array}{rlrl}
Acq'_1.p \mathrel{\widehat{=}} & \ulcorner x \bullet [\![\neg x]\!] \lrcorner_{\Phi} \ ; [\![\neg x]\!]^* \ ; [\![x]\!] \ ; Acq.p & Acq'_2.p \mathrel{\widehat{=}} & \ulcorner x \bullet [\![x]\!] \ ; (x := 0) \lrcorner_{\Phi} \\
TryOK'.p \mathrel{\widehat{=}} & \ulcorner x \bullet [\![x]\!] \ ; (x := 0) \lrcorner_{\Phi} \ ; (r_p := true) & TryFl'.p \mathrel{\widehat{=}} & \ulcorner x \bullet [\![\neg x]\!] \lrcorner_{\Phi} \ ; (r_p := false) \\
Exec'.p \mathrel{\widehat{=}} & \multicolumn{3}{l}{((Acq'_1 \ ; \mathsf{id} \ ; Rel) \vee (Acq'_2 \ ; \mathsf{id} \ ; Rel) \vee (TryOK' \ ; \mathsf{id} \ ; Rel) \vee TryFl').p} \\
Prog'.P \mathrel{\widehat{=}} & \multicolumn{3}{l}{\{x\} \bigwedge_{p:P} \left(\mathsf{id} \ ; \left(\{\mathrm{B}_p = \langle\,\rangle\} Exec' \ ; post_{\Phi}.x.(\widetilde{\Phi}^{+})\right)^{\omega}\right).p}
\end{array} \qquad \square$$

Clearly, transforming *Prog* to *Prog′* by reasoning at trace-based level of Definition 1 is infeasible. Therefore, we develop a number of refinement laws that are applied to our example. First, we have the following; the proof of each equivalence is straightforward.

Law 2 *If $p \in Proc$, C and D are commands, each C_i is a command and $\overline{v}$ is a vector of variables, then*

$$\boxed{\overline{v} \bullet C}_{\Phi} \mathrel{\underline{\square}}_p pre_{\Phi}.\overline{v}.\ulcorner \overline{v} \bullet C \lrcorner_{\Phi} \tag{4}$$

$$pre_{\Phi}.C \mathrel{\underline{\square}}_p pre_{\Phi}.(\{\mathrm{B}_p = \langle\,\rangle\}C) \tag{5}$$

$$pre_{\Phi}.\overline{v}.\ulcorner \overline{v} \bullet C \lrcorner_{\Phi} \ ; D \mathrel{\underline{\square}}_p pre_{\Phi}.\overline{v}.\left(\ulcorner \overline{v} \bullet C \lrcorner_{\Phi} \ ; D\right) \qquad \text{provided } C.p \Rrightarrow \neg\varepsilon \tag{6}$$

$$\textstyle\bigvee_i pre_{\Phi}.C_i \mathrel{\underline{\square}}_p pre_{\Phi}.\left(\bigvee_i C_i\right) \tag{7}$$

$$C \ ; pre_{\Phi}.D \mathrel{\underline{\square}}_p post_{\Phi}.C \ ; D \qquad \text{provided } C.p \vee D.p \Rrightarrow \neg\varepsilon \tag{8}$$

To transform *Prog* to *Prog′*, we develop a leapfrog theorem analogous to Law 1, whose proof uses the equivalences defined in Law 2 as well as $PF.\overline{v} \mathrel{\widehat{=}} post_{\Phi}.\overline{v}.\widetilde{\Phi}^{+}$.

Theorem 1 (Leapfrog flush). *Suppose $p \in Proc$, each C_i and D_i is a command such that $C_i.p \Rrightarrow \neg\varepsilon$, and $\overline{v}$ is a vector of variables. Then*

$$\left(\widetilde{\Phi}^{+} \ ; \left(\textstyle\bigvee_i \boxed{\overline{v} \bullet C_i}_{\Phi} \ ; D_i\right)\right)^{\omega} ; PF.\overline{v} \ \ \mathrel{\underline{\square}}_p \ \ PF.\overline{v} \ ; \left(\{\mathrm{B}_p = \langle\,\rangle\} \left(\textstyle\bigvee_i \ulcorner \overline{v} \bullet C_i \lrcorner_{\Phi} \ ; D_i\right) ; PF.\overline{v}\right)^{\omega}$$

The left hand side of Theorem 1 contains a disjunction that executes $\boxed{\overline{v} \bullet C_i}_{\Phi}$, which ensures the buffer is empty (via flushes if necessary) both before and after execution of C_i. After the end of the iteration, command $PF.\overline{v}$ is executed, which ensures the buffer is empty when the process terminates; flushes may be necessary due to the behaviour of D_i. On the right hand side, each iteration is guaranteed to start with an empty buffer and each disjunct starts with the weaker $\ulcorner \overline{v} \bullet C_i \lrcorner_{\Phi}$, which only flushes the buffer at the end of execution. However, each iteration ends with $PF.\overline{v}$. Further note that on the right hand side, each iteration is guaranteed to begin in a state where the buffer of p is empty.

Proposition 1. *$Prog \sqsupseteq_P Prog'$*

Proof. Applying Theorem 1 to *Prog*, we obtain:

$$Prog''.P \ \mathrel{\widehat{=}} \ \{x\} \bigwedge_{p:P} \left(\{\mathrm{B}_p = \langle\,\rangle\} PF.x \ ; \left(\{\mathrm{B}_p = \langle\,\rangle\} Exec' \ ; PF.x\right)^{\omega}\right).p$$

Then, because $\{\mathrm{B}_p = \langle\,\rangle\} PF.x \sqsubseteq_p \mathsf{id}$, we have $Prog'' \sqsubseteq_P Prog'$. □

For the proof in Section 5.3, we find the following laws to be useful, each of which is proved in a straightforward manner. Note that for (11) and (14), the refinement only holds in one direction. Of these, (14) states that an assignment $\{\mathrm{B}_p = \langle\,\rangle\} v := k$ either ends with $\mathrm{B}_p = \langle (v, k) \rangle$, or the buffer B_p is flushed as part of the assignment semantics.

Law 3 *Suppose C and D are commands, p is a process, v is a variable and k a value. Then each of the following holds:*

$$\{\mathrm{B}_p = \langle\,\rangle\}\boxed{\overline{v} \bullet C}_{\Phi} \ \sqsubseteq_p \{\mathrm{B}_p = \langle\,\rangle\}\bbox[border:1px dashed]{\overline{v} \bullet C}_{\Phi} \tag{9}$$

$$\bbox[border:1px dashed]{\overline{v} \bullet C\,;\,D}_{\Phi} \ \sqsubseteq_p \boxed{\overline{v} \bullet C}\ ;\ \bbox[border:1px dashed]{\overline{v} \bullet D}_{\Phi} \quad \textit{provided } C.p \vee D.p \Rrightarrow \neg\varepsilon \tag{10}$$

$$\{\mathrm{B}_p = \langle\,\rangle\}\bbox[border:1px dashed]{v \bullet v := k}_{\Phi} \ \sqsupseteq_p \boxed{v \bullet \mathsf{id}\,;\, v \Mapsfrom k\,;\, \mathsf{id}}\,\{\mathrm{B}_p = \langle\,\rangle\} \tag{11}$$

$$\boxed{\overline{v} \bullet C}\ ;\ \boxed{\overline{v} \bullet D} \ \sqsubseteq_p \boxed{\overline{v} \bullet C\,;\,D} \tag{12}$$

$$\{\mathrm{B}_p = \langle\,\rangle\}\boxed{\overline{v} \bullet [\![b]\!]} \ \sqsubseteq_p \{\mathrm{B}_p = \langle\,\rangle\}\boxed{\overline{v} \bullet [b]} \tag{13}$$

$$\{\mathrm{B}_p = \langle\,\rangle\}v := k \sqsupseteq_p (\mathsf{id}\,;\, v \mapsfrom k\,;\, \mathsf{id})\{\mathrm{B}_p = \langle(v,k)\rangle\}\ ;\ (\varepsilon \vee (\Phi\,;\mathsf{id})) \tag{14}$$

5.2 Local Buffer Refinement

In this section, we develop a novel method for proving refinement for TSO architectures, where buffer effects are taken into account. The method allows one to prove that a (concrete) process with buffer effects has the same behaviour as an (abstract) process without the buffer. In essence, one may pretend that the effect of local changes have already been flushed at the abstract level. This is allowed because in the context of the overall behaviour of a program, it makes no difference whether a variable local to process p has a pending write in p's local buffer, or in shared memory. This essentially removes the potential non-determinism that arises from reasoning about flushes for local variables. We let

$$\begin{aligned} \mathit{LCover}.P.(\sigma, \pi) &\mathrel{\widehat{=}} \big(\sigma \oplus \textstyle\bigcup_{p:P} \mathit{LVar}.p \lhd \mathit{cover}.(\sigma.\mathrm{B}_p),\ \pi\big) \\ \mathit{LBuffer}.P.s &\mathrel{\widehat{=}} \lambda\, t{:}\,\mathbb{Z} \bullet \mathit{LCover}.P.(s.t) \end{aligned}$$

Here $\mathit{cover}.(\sigma.\mathrm{B}_p)$ generates a set of pairs from $\sigma.\mathrm{B}_p$ and $\mathit{LVar}.p \lhd \mathit{cover}.(\sigma.\mathrm{B}_p)$ restricts these to the local variables of p. Within $\mathit{LCover}.P.(\sigma, \pi)$ such a localised mapping is generated for each $p \in P$, and then, σ is overwritten by this mapping.

Definition 2. *If C and A are commands, and P is a non-empty set of processes, we say C* buffer refines *A* with respect to *P, (denoted $C \Supset_P A$) iff for any interval Δ and stream s, $(C.P).\Delta.s \Rightarrow (A.P).\Delta.(\mathit{LBuffer}.P.s)$.*

Thus, whenever C holds for a set of processes P over interval Δ in stream s, command A must hold for P in Δ for the masked stream $\mathit{LBuffer}.P.s$. In particular, this implies that A behaves as if the local buffer effects of the concrete command have already been applied. We write $\Supset_p$ for $\Supset_{\{p\}}$.

Clearly, reasoning at the level of Definition 2 is infeasible. Instead, we explore some higher level properties for the programming constructs we use. In general, buffer refinement is neither reflexive nor transitive[8], e.g., if C always reads from main memory regardless of the state of the buffer, then reflexivity does not hold. However, $\Supset_P$ may be combined with standard refinement as follows, which holds trivially after expanding the definitions of $\sqsupseteq_P$ and $\Supset_P$.

Theorem 2. *If $C' \sqsupseteq_P C$, $C \Supset_P A$ and $A \sqsupseteq_P A'$ then $C' \Supset_P A'$.*

[8] Fully exploring the properties of $\Supset_P$ lie outside the scope of this paper.

We immediately have the following laws, which help simplify verification of assignments and buffer flushes.

Law 4 *If $v \in Var$, $p \in Proc$, $k \in Val$, $P \mathrel{\widehat{=}} \mathrm{B}_p \neq \langle\,\rangle \wedge (\mathrm{B}_p.0 \upharpoonright 1) \in LVar.p$, and $Q \mathrel{\widehat{=}} \mathrm{B}_p \neq \langle\,\rangle \wedge (\mathrm{B}_p.0 \upharpoonright 1) \notin LVar.p$, then*

$$\{v \in LVar.p\}v \mapsfrom k \Supset_p v \Mapsfrom k \quad (15) \qquad \{P\}\,\Phi \Supset_p \mathsf{id} \quad (17)$$

$$\{v \notin LVar.p\}v \mapsfrom k \Supset_p \mathsf{id} \quad (16) \qquad \{Q\}\,\Phi \Supset_p (\mathrm{B}_p.0 \upharpoonright 1) \Mapsfrom (\mathrm{B}_p.0 \upharpoonright 2) \quad (18)$$

By conditions (15) and (16), adding a pending write (v, k) to p's buffer is a local buffer refinement of a global update to v whenever $v \in LVar.p$, and of id, otherwise. On the other hand, (17) states that flushing a local variable of process p from p's buffer has the same effect as executing id abstractly (because the effect of the variable has already occurred), and condition (18) states that flushing a global variable has the same effect as executing the corresponding write in memory at the abstract level.

The laws below allow local buffer refinement to be decomposed, and the pre/post assertions under local buffer refinement to strengthened.

Law 5 *If $C \Supset_P A$, $C_1 \Supset_P A_1$ and $C_2 \Supset_P A_2$ for a set of processes P, and b, c are state predicates such that $b \Rightarrow c$ and $(vars.b \cup vars.c) \cap (\bigcup_{p:P} LVar.p) = \varnothing$, then*

$$C_1 \,;\, C_2 \Supset_P A_1 \,;\, A_2 \quad (19) \qquad C_1 \vee C_2 \Supset_P A_1 \vee A_2 \quad (22)$$

$$C^\omega \Supset_P A^\omega \quad (20) \qquad \{b\}C \Supset_P \{c\}A \quad (23)$$

$$C_1 \wedge C_2 \Supset_P A_1 \wedge A_2 \quad (21) \qquad C\{b\} \Supset_P A\{c\} \quad (24)$$

Law 6 (Parallel composition) *If $C'.P \mathrel{\widehat{=}} \bigwedge_{q:P} C.q$, $A'.P \mathrel{\widehat{=}} \bigwedge_{q:P} A.q$ and $C \Supset_q A$ holds for each for each $q \in P$, then $C' \Supset_P A'$.*

5.3 Application: Spinlock Example

We now apply our rules to the running spinlock example (modelled by *Prog*), and prove it to be a refinement of the abstract program (modelled by *Spec*). Our notion of refinement is local buffer refinement (Definition 2), i.e., we show

$$Prog \Supset_P Spec \tag{25}$$

for an arbitrarily chosen non-empty set of processes P. Various refinement rules may be introduced to generalise the theory as needed. By Theorem 2 and Proposition 1, (25) immediately reduces to a proof of $Prog' \Supset_P Spec$. Using (23), followed by Law 6, proof of $Prog' \Supset_P Spec$ again reduces to the following for some arbitrarily chosen $p \in P$.

$$\mathsf{id} \,;\, \Big(\{\mathrm{B}_p = \langle\,\rangle\}Exec' \,;\, post_\Phi.x.(\widetilde{\Phi}^+)\Big)^\omega \Supset_p (\mathsf{id} \,;\, AExec)^\omega \,;\, \mathsf{id} \tag{26}$$

We use $\mathsf{id} \sqsubseteq_p \mathsf{id} \,;\, \mathsf{id}$ to split the first id on the right hand side of (26), then Law 1, to obtain $\mathsf{id} \,;\, (\mathsf{id} \,;\, AExec.p \,;\, \mathsf{id})^\omega$. Hence, using (19) followed by (20), the proof of (26) reduces again to a proof of

$$\{\mathrm{B}_p = \langle\,\rangle\}Exec' \,;\, post_\Phi.x.(\widetilde{\Phi}^+) \Supset_p \mathsf{id} \,;\, AExec \,;\, \mathsf{id} \tag{27}$$

Then using (22), condition (2) and the fact that ; distributes over $\vee$, we are left with a number of proof obligations for each disjunct of $Exec'$. Of these, we present the most complex: the proof obligation for Acq_1'.

$$\{\mathrm{B}_p = \langle\,\rangle\}Acq_1' \,;\, \mathsf{id} \,;\, Rel \,;\, post_{\Phi}.x.(\widetilde{\Phi}^{+}) \Supset_p \mathsf{id} \,;\, AAcq \,;\, \mathsf{id} \,;\, ARel \,;\, \mathsf{id} \tag{28}$$

It is trivial to show $\{\mathrm{B}_p = \langle\,\rangle\}\begin{array}{:c:}\hdashline x \bullet [\![\neg x]\!] \\ \hdashline\end{array}_{\Phi} \,;\, [\![\neg x]\!]^{*} \,;\, [\![x]\!] \,;\, Acq \sqsupseteq_p \mathsf{id} \,;\, \{\mathrm{B}_p = \langle\,\rangle\}Acq$, and using $\mathsf{id} \sqsubseteq_p \mathsf{id} \,;\, \mathsf{id}$ the proof of (28) reduces as follows, where the initial part of Acq_1' is refined to id.

$$\{\mathrm{B}_p = \langle\,\rangle\}Acq \,;\, \mathsf{id} \,;\, Rel \,;\, post_{\Phi}.x.(\widetilde{\Phi}^{+}) \Supset_p \mathsf{id} \,;\, AAcq \,;\, \mathsf{id} \,;\, ARel \,;\, \mathsf{id} \tag{29}$$

We now focus on the initial part of the left hand side, where we distinguish between $ALoop \mathrel{\widehat{=}} \boxed{x \bullet [\![\neg x]\!]}_{\Phi} \,;\, [\![\neg x]\!]^{\omega} \,;\, [\![x]\!]$ and $ADo \mathrel{\widehat{=}} \boxed{x \bullet [\![x]\!] \,;\, (x := 0)}_{\Phi}$.

$$\begin{array}{ll}
\quad \{\mathrm{B}_p = \langle\,\rangle\}Acq & \\
\sqsubseteq_p \{\mathrm{B}_p = \langle\,\rangle\}ADo \vee \{\mathrm{B}_p = \langle\,\rangle\}ALoop^{\omega+} \,;\, ADo & \text{defn of } Acq\text{, then } g^{\omega} \equiv \varepsilon \vee g^{\omega+}
\end{array}$$

Using $\{\mathrm{B}_p = \langle\,\rangle\}ALoop^{\omega+} \Supset_p \mathsf{id}\{\mathrm{B}_p = \langle\,\rangle\}$ (proof elided) condition (29) reduces to

$$\{\mathrm{B}_p = \langle\,\rangle\}ADo \,;\, \mathsf{id} \,;\, Rel \,;\, post_{\Phi}.x.(\widetilde{\Phi}^{+}) \Supset_p AAcq \,;\, \mathsf{id} \,;\, ARel \,;\, \mathsf{id} \tag{30}$$

Again focusing on the initial part of the left hand side, we have:

$$\begin{array}{ll}
\quad \{\mathrm{B}_p = \langle\,\rangle\}\ ADo & \\
\sqsubseteq_p \{\mathrm{B}_p = \langle\,\rangle\}\begin{array}{:c:}\hdashline x \bullet [\![x]\!] \,;\, (x := 0) \\ \hdashline\end{array}_{\Phi} & \text{defn of } ADo \text{ and by (9)} \\
\sqsubseteq_p \{\mathrm{B}_p = \langle\,\rangle\}\boxed{x \bullet [\![x]\!]} \,;\, \begin{array}{:c:}\hdashline x \bullet (x := 0) \\ \hdashline\end{array}_{\Phi} & \text{by (10)} \\
\sqsupseteq_p \{\mathrm{B}_p = \langle\,\rangle\}\boxed{x \bullet [\![x]\!]} \,;\, \{\mathrm{B}_p = \langle\,\rangle\}\begin{array}{:c:}\hdashline x \bullet (x := 0) \\ \hdashline\end{array}_{\Phi} & \text{by logic} \\
\sqsupseteq_p \{\mathrm{B}_p = \langle\,\rangle\}\boxed{x \bullet [\![x]\!]} \,;\, \boxed{x \bullet \mathsf{id} \,;\, (x \Lleftarrow 0) \,;\, \mathsf{id}}\{\mathrm{B}_p = \langle\,\rangle\} & \text{by (11)} \\
\sqsubseteq_p \{\mathrm{B}_p = \langle\,\rangle\}\boxed{x \bullet [\![x]\!] \,;\, \mathsf{id}} \,;\, \boxed{x \bullet (x \Lleftarrow 0) \,;\, \mathsf{id}}\{\mathrm{B}_p = \langle\,\rangle\} & \text{using (12) twice} \\
\sqsupseteq_p \{\mathrm{B}_p = \langle\,\rangle\}\boxed{x \bullet [\![x]\!]} \,;\, \boxed{x \bullet (x \Lleftarrow 0) \,;\, \mathsf{id}}\{\mathrm{B}_p = \langle\,\rangle\} & [\![x]\!] \,;\, \mathsf{id} \sqsubseteq_p [\![x]\!] \\
\sqsupseteq_p \{\mathrm{B}_p = \langle\,\rangle\}\boxed{x \bullet [x]} \,;\, \boxed{x \bullet (x \Lleftarrow 0) \,;\, \mathsf{id}}\{\mathrm{B}_p = \langle\,\rangle\} & \text{by (13)} \\
\sqsubseteq_p \boxed{x \bullet [x] \,;\, (x \Lleftarrow 0) \,;\, \mathsf{id}}\{\mathrm{B}_p = \langle\,\rangle\} & \text{by (12) and weakening}
\end{array}$$

It is straightforward to show $\boxed{x \bullet [x](x \Lleftarrow 0) \,;\, \mathsf{id}} \Supset_p AAcq \,;\, \mathsf{id}$, which expanding Rel and using (14) further reduces (30) to a proof of:

$$(\mathsf{id} \,;\, x \hookleftarrow 1 \,;\, \mathsf{id})\{\mathrm{B}_p = \langle(x,1)\rangle\} \,;\, (\varepsilon \vee \Phi \,;\mathsf{id}) \,;\, post_{\Phi}.x.(\widetilde{\Phi}^{+}) \Supset_p \mathsf{id} \,;\, ARel \,;\, \mathsf{id}$$

Using $\mathsf{id} \sqsubseteq_p \mathsf{id} \,;\, \mathsf{id}$ the right hand side transforms to $\mathsf{id} \,;\, \mathsf{id} \,;\, ARel \,;\, \mathsf{id}$. Then, using (16) and Theorem 2, we have $\mathsf{id} \,;\, x \hookleftarrow 1 \,;\, \mathsf{id} \Supset_p \mathsf{id}$, and hence, using (19), we obtain $\{\mathrm{B}_p = \langle(x,1)\rangle\}(post_{\Phi}.x.(\widetilde{\Phi}^{+}) \vee (\Phi \,;\mathsf{id} \,;\, post_{\Phi}.x.(\widetilde{\Phi}^{+})) \Supset_p \mathsf{id} \,;\, ARel \,;\, \mathsf{id}$. Finally, because $\Phi \,;\mathsf{id} \,;\, post_{\Phi}.x.(\widetilde{\Phi}^{+}) \sqsupseteq_p post_{\Phi}.x.(\widetilde{\Phi}^{+})$, we are left with

$$\{\mathrm{B}_p = \langle(x,1)\rangle\}post_{\Phi}.x.(\widetilde{\Phi}^{+}) \Supset_p \mathsf{id} \,;\, ARel \,;\, \mathsf{id} \tag{31}$$

Because there is exactly one item in the buffer, and because $post_{\Phi}.x.(\widetilde{\Phi}^{+})$ guarantees that a flush occurs, the left hand side reduces to $\{\mathrm{B}_p = \langle(x,1)\rangle\}\mathsf{id} \,;\, \Phi \,;\mathsf{id}$. Finally, using the fact that B_p is a local variable of p and is not modified by id followed by (18), our proof is completed.

Notable in our verification is that concurrency aspects hardly need to be considered. The fact that the locking mechanisms guarantee safety is understood at the level of *Spec*. The refinement proof only requires consideration of the local buffer. This is in contrast to existing methods which require global conditions to be checked, e.g., [20] checks race conditions, [7, 16, 25] check linearizability, and [9] checks reduction. We conjecture that more complex examples will indeed require consideration of the behaviour of other processes. To this end, we will integrate compositional methods such as rely/guarantee into our framework [13, 11].

6 Conclusions

Existing approaches to relaxed memory verification (e.g., [6, 21, 22, 9, 20, 7, 16]) focus on a low-level language (i.e., individual reads/writes), and hence, to perform a verification, programs need to be observed and understood in their (verbose) low-level representation. We are not aware of any approach that tries to lift memory model effects to a higher level of abstraction; our work here is hence unique in this sense [15].

The basic idea is to think of a statements as being executed over an interval of time or an execution window. Such execution windows can overlap if programs are executed concurrently and overlapping windows correspond to program instructions that can be executed in any order, representing the effect of concurrent executions and reorderings due to TSO. Overlapping execution windows may also interfere with each other and fixing the outcome of an execution within a window can influence the outcome within another. This paper presents several advances to the semantics in [15] by simplifying the interval logic, and program semantics, as well as developing buffer-specific rules for expression evaluation and refinement. The underlying rules are algebraic in nature, and hence, we provide generic transformation laws, which are in turn applied to our running example.

A difficulty when reasoning about TSO memory is that in addition to the normal non-determinism caused by concurrency, an additional level of non-determinism is introduced via use of local buffers. The methods in this paper allow one to reduce the non-determinism that must be considered when reasoning about local updates. In particular, we develop a notion of *local buffer refinement*, which allows one to proceed as if pending writes to local variables have already occurred in the abstract level. In particular, this means that local writes do not appear out of order. A similar observation is used for local transformation in the context of compilers for weak memory [8], however, these do not consider higher-level synchronisation instructions such as `lock`.

As part of future work, we aim to study the connections between local buffer refinement, and existing notions such as triangular race freedom [20] and reduction [9].

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
2. J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012.
3. R. J. R. Back and J. von Wright. Reasoning algebraically about loops. *Acta Informatica*, 36(4):295–334, July 1999.
4. D. Bovet and M. Cesati. *Understanding the Linux Kernel*. OReilly, 3rd edition, 2005.

5. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
6. S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.
7. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In H. Seidl, editor, *ESOP*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.
8. S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In R. Gupta, editor, *CC*, volume 6011 of *LNCS*, pages 104–123. Springer, 2010.
9. E. Cohen and B. Schirmer. From total store order to sequential consistency: A practical reduction theorem. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *LNCS*, pages 403–418. Springer, 2010.
10. B. Dongol and J. Derrick. Data refinement for true concurrency. In J. Derrick, E. A. Boiten, and S. Reeves, editors, *Refine*, volume 115 of *EPTCS*, pages 15–35, 2013.
11. B. Dongol, J. Derrick, and I. J. Hayes. Fractional permissions and non-deterministic evaluators in interval temporal logic. *ECEASST*, 53, 2012.
12. B. Dongol and I. J. Hayes. Deriving real-time action systems in a sampling logic. *Sci. Comput. Program.*, 78(11):2047–2063, 2013.
13. B. Dongol, I. J. Hayes, and J. Derrick. Deriving real-time action systems with multiple time bands using algebraic reasoning. *Sci. of Comp. Prog.*, 85, Part B(0):137 – 165, 2014.
14. B. Dongol, I. J. Hayes, L. Meinicke, and K. Solin. Towards an algebra for real-time programs. In W. Kahl and T.G. Griffin, editors, *RAMiCS*, volume 7560 of *LNCS*, pages 50–65, 2012.
15. B. Dongol, O. Travkin, J. Derrick, and H. Wehrheim. A high-level semantics for program execution under total store order memory. In Z. Liu, J. Woodcock, and H. Zhu, editors, *ICTAC*, volume 8049 of *LNCS*, pages 177–194. Springer, 2013.
16. A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In M. K. Aguilera, editor, *DISC*, volume 7611 of *LNCS*, pages 31–45. Springer, 2012.
17. I. J. Hayes, A. Burns, B. Dongol, and C. B. Jones. Comparing degrees of non-determinism in expression evaluation. *Comput. J.*, 56(6):741–755, 2013.
18. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
19. B. C. Moszkowski. A complete axiomatization of Interval Temporal Logic with infinite time. In *LICS*, pages 241–252, 2000.
20. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.
21. S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *SPAA*, pages 34–41, 1995.
22. 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.
23. 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, 2011.
24. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.
25. O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC'13*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.