An ACL2 Mechanization of an Axiomatic Framework for Weak Memory

Benjamin Selfridge
University of Texas at Austin
Austin, TX
benself@cs.utexas.edu

Proving the correctness of programs written for multiple processors is a challenging problem, due in no small part to the weaker memory guarantees afforded by most modern architectures. In particular, the existence of store buffers means that the programmer can no longer assume that writes to different locations become visible to all processors in the same order. However, all practical architectures do provide a collection of weaker guarantees about memory consistency across processors, which enable the programmer to write provably correct programs in spite of a lack of full sequential consistency. In this work, we present a mechanization in the ACL2 theorem prover of an axiomatic weak memory model (introduced by Alglave et al. [2]). In the process, we provide a new proof of an established theorem involving these axioms.

1 Introduction

Analysis of sequential programs is a well-understood problem for which a variety of proof techniques and methodologies exist. [5] Many of these techniques can be adapted to a multiprocessor setting if we assume *sequential consistency* (SC) - i.e., that for any concurrent execution of the program, there exists an interleaving of the memory events that is consistent with both the program order and the communication dependencies between processes. [6, 8] However, sequential consistency turns out to be a much stronger requirement than is practically necessary. Moreover, due to the inherently high runtime and resource penalties of SC, designers of multiprocessor architectures are motivated to relax this constraint in order to achieve better performance.

To understand why a lack of sequential consistency impacts us as programmers, consider the following example. Suppose our architecture consists of a number of processors $P_1, \ldots, P_n$ and a shared memory M. Assume that when a processor issues a write to memory, that write is immediately visible to all other processors.

Consider the program execution represented in Figure 1. Each processor assigns the value 1 to memory location x or y, and reads the value at the other location into a register. (Assume x and y are both initially equal to 0.) Now, we ask the question: what are the possible values of registers r_0 and r_1 after running this program? It is easy to see that $r_0 = 1$, $r_1 = 1$ is one possible final state, obtained by a scheduler that alternates between P_0 and P_1. We can also obtain $r_0 = 0$, $r_1 = 1$ by running P_0's program to the end, and then subsequently running P_1's program to the end. Likewise, it is also possible to obtain $r_0 = 1$, $r_1 = 0$. These are the only possible final states, because this (sketch of an) architecture is sequentially consistent; every processor completely executes its first instruction before continuing to the second.

Now, consider the following modification of this architecture. Each of the processors P_i is equipped with a *store buffer* B_i. When P_i issues a write, instead of propagating the write directly to shared memory, the write is initially sent to buffer B_i. That write will eventually hit memory, although we have no

F. Verbeek and J. Schmaltz (Eds.): ACL2 Workshop 2014 (ACL2'14).
EPTCS 152, 2014, pp. 129–144, doi:10.4204/EPTCS.152.11

P_0	P_1
$x \leftarrow 1$	$y \leftarrow 1$
$r_0 \leftarrow y$	$r_1 \leftarrow x$

Figure 1: A multiprocessor program execution. The final state $r_0 = 0$, $r_1 = 0$ is prohibited by sequential consistency, but is possible on an architecture with store buffers.

guarantee of when that will happen (unless the programmer inserts an explicit memory fence). If P_i wishes to read a value from memory, it first checks its own store buffer to see if it has issued any pending writes to that memory location. If it has, it uses that value; otherwise, it obtains the value from memory.

If we run the same program on this architecture, it is easy to see that the final state $r_0 = 0$, $r_1 = 0$ is obtainable if neither processor's store buffer is flushed before the reads are performed; both processors issue a write, but those writes are not globally visible by the time each process issues its read, and hence both processors read the "old" values of x and y. This is a clear violation of sequential consistency. There is no way to linearly order the instructions of the two programs as atomic memory events and obtain this final state; nevertheless, this behavior is possible on this architecture. This odd behavior isn't merely a theoretical possibility; it is actually observable on x86 machines.

In spite of the fact that we do not generally have sequential consistency, most weaker memory models do uphold a set of guarantees which, though they are not as strong as sequential consistency, do prohibit certain behaviors. These guarantees vary greatly from model to model [3, 4, 7, 9, 10], and the variety and abundance of these models suggests the need for a more generic framework for weak memory. Such a framework ought to be both general enough to capture the semantics of all modern architectures, and strong enough to enforce meaningful constraints that are universally upheld. One such framework is introduced in Alglave et al. [2], and in this paper we present its mechanization in ACL2. Furthermore, we present a new proof of an established theorem about this framework, and we discuss the mechanized proof.

A brief notational remark: throughout this paper, given a relation R, we will let R^+ denote the irreflexive transitive closure of R. Given two relations R and Q, we let $R;Q$ denote the sequencing of R and Q, i.e.

$$x \xrightarrow{R;Q} y \text{ iff. } \exists p,\ x \xrightarrow{R} p \xrightarrow{Q} y.$$

2 Background: An Axiomatic Framework for Weak Memory

The execution of a sequential program results in a linear sequence of events (usually reads or writes from/to a location in memory). The event order derived from this sequence is called the *program order*. The program order is a total order on all events, and from this order we can reason in a straightforward way about the possible final states that can result from a run of the program by considering all possible event orderings and demonstrating that they all produce a final state in a particular configuration.

With concurrent programs, however, the situation is more complicated. Generally speaking, an execution on a concurrent machine is not simply a sequence of events with a global program order. Events that occur on different processors are not necessarily comparable, because a write issued by one processor may not be visible to any other processors for some time (despite being immediately visible to the process that executed it). Therefore, in order to specify a set of requirements for our weaker memory guarantees, we need a weakened definition of a program execution that retains enough structure to be amenable to subsequent constraints and analyses. In this section, we describe a compelling axiomatic framework for weak memory [2], which includes both a more general notion of execution for multiple processors and a parameterized set of requirements that is meant to characterize all modern multiprocessor architectures.

2.1 Concurrent Executions

We begin with two definitions.

Definition. An *event e* is an object which consists of a unique identifier $\mathsf{id}(e)$, a process $\mathsf{proc}(e)$, a type $\mathsf{type}(e)$ which identifies e as being either a read or a write, an address $\mathsf{addr}(e)$ equal to the address in memory that e reads from or writes to, and a value $\mathsf{val}(e)$ equal to the value read or written by e.

Definition. An *execution* is a tuple $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ where $\mathbb{E}$ is a collection of events, and po, co, and rf are all relations on $\mathbb{E}$ satisfying:

- po is a total order on events, when restricted to a single process
- co is a total order on writes, when restricted to a single address
- rf is a relation from writes to reads such that for all reads $r \in \mathbb{E}$, there exists a unique write $w \in \mathbb{E}$ such that $w \xrightarrow{\mathsf{rf}} r$ (we also require that $\mathsf{val}(w) = \mathsf{val}(r)$).

The relation po is undefined on events belonging to different processes, and likewise, co is undefined on any pair of events that are not writes to the same address.

The relation po is our concurrent version of program order; it is a total order not on all events, but only on those belonging to the same processor. The "coherence order" co is a total order on writes to the same location in memory. This order corresponds to our intuition that the writes to each individual location hit memory in a particular sequential order. The read-from relation rf captures the dependency between writes and reads; $w \xrightarrow{\mathsf{rf}} r$ means "r takes its value from the write w." [1] It is a surjective relation with a one-sided inverse function, rf^{-1}.

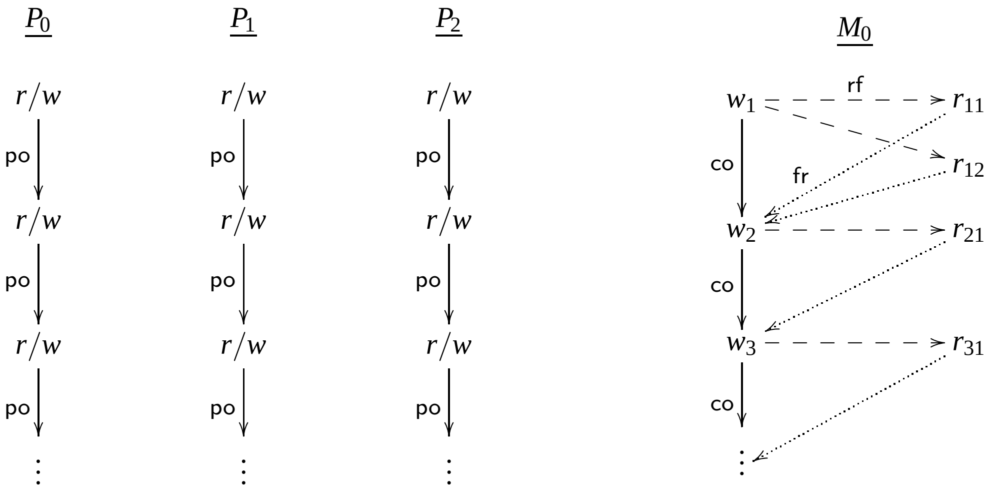
The purpose of co and rf is to capture interprocess dependencies between events occurring at the same location; co captures dependencies between two writes arising from their relative visibility with respect to time, and rf captures the dependency of reads on the writes they take their value from. However, it is also intuitively possible to have a write "depend" on a read. If w, w' are writes and r is a read such that $w' \xrightarrow{\mathsf{rf}} r$ and $w' \xrightarrow{\mathsf{co}} w$, then there is a sense in which w "comes after" r, because r takes its value from an earlier write. Therefore, we have another relation, which we refer to as the "from-read" relation.

Definition. Let $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ be an execution. The "from-read" relation fr is defined as

$$\mathsf{fr} = \mathsf{rf}^{-1}; \mathsf{co},$$

i.e. $r \xrightarrow{\mathsf{fr}} w$ if there exists a write w' such that $w' \xrightarrow{\mathsf{rf}} r$ and $w' \xrightarrow{\mathsf{co}} w$. (Note that this is equivalent to stating that $\mathsf{rf}^{-1}(r) \xrightarrow{\mathsf{co}} w$.)

[1] The reader may be wondering why we choose to write $w \xrightarrow{\mathsf{rf}} r$ rather than $r \xrightarrow{\mathsf{rf}} w$ - the latter certainly seems more sensible when read aloud ("r read-from w"). The reason is that the direction of the arrow is meant to represent a dependency between two events, with the arrow pointing toward the dependent ("later") event. This will enable us to state our weak memory requirements as assertions of the acyclicity of various combinations of these and other relations.



(a) The per-process view.

(b) The per-location view.

Figure 2: Two views of memory events. In figure (b), solid lines are co, dashed lines are rf, and dotted lines are fr. For po, co and fr, not all arrows are pictured, as po and co are transitively closed.

Our three relations rf, co, and fr will be sufficient to specify certain communication dependencies regarding reads and writes to the same location. We abbreviate the three into a single relation.

Definition. Let $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ be an execution. The relation com is defined as

$$\mathsf{com} = \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr},$$

i.e. $x \xrightarrow{\mathsf{com}} y$ if $x \xrightarrow{\mathsf{co}} y$, $x \xrightarrow{\mathsf{rf}} y$, or $x \xrightarrow{\mathsf{fr}} y$.

The po and com relations represent two distinct types of dependencies between events; po captures *per-process* dependencies, and com relation captures *per-location* dependencies. The existence of these two relations suggests two distinct views of our event graph. The first is the per-process view, where we organize all the events by the process they belong to, and list them in program order (see Figure 2a). The second is the per-location view, where we organize the events by the memory location at which they occur, and list each write event in coherence order (see Figure 2b for an example of what this might look like for a particular location M_0).

2.2 Sequential Consistency and SC-Per-Location

In the previous section, we presented a generalization of the notion of a sequential execution to an arbitrary number of processors. Whereas a sequential execution has a single relation, the program order (which is a total order on all events), a concurrent execution consists of two: its per-process program order po, and the communication dependency relation com. In our framework, the usual definition of sequential consistency [6] is that there exists a completion of the relation $\mathsf{po} \cup \mathsf{com}$ which is a total order on all events. An equivalent way to state this is that the relation $\mathsf{po} \cup \mathsf{com}$ is acyclic, and so we have the following definition:

Definition. An execution $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ is *sequentially consistent* (SC) if

$$\mathsf{acyclic}(\mathsf{po} \cup \mathsf{com}),$$

i.e. the union of the po and com relations is acyclic.

As we have already discussed, sequential consistency does not hold in general for modern multiprocessor architectures. However, if we restrict the program order po to events at the same location, then we get a new, weaker property. As it happens, this property holds for all modern architectures.

To this end, we define another relation, pol, which is the restriction of po to events that occur at the same location.

Definition. Let $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ be an execution. The relation pol is defined as

$$\mathsf{pol} = \{(x,y) \in \mathbb{E} \times \mathbb{E} \mid x \xrightarrow{\mathsf{po}} y \text{ and } \mathsf{addr}(x) = \mathsf{addr}(y)\},$$

i.e. $x \xrightarrow{\mathsf{pol}} y$ if $x \xrightarrow{\mathsf{po}} y$ and x and y have the same address.

We are now in a position to reproduce the definition for a weakened version of sequential consistency for concurrent executions (originally given in [2]), which we refer to as sequential consistency per location.

Definition. An execution $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ is *sequentially consistent per location* (SC-Per-Location) if

$$\mathsf{acyclic}(\mathsf{pol} \cup \mathsf{com}),$$

i.e. the union of the pol and com relations is acyclic.

The intuition behind this definition is that if we restrict ourselves to examining one memory location, the system appears to be sequentially consistent. The acyclicity of program order and the communication relations co, rf and fr guarantee the existence of a sequential execution of these events that produces the same behavior (for *this* memory location) as the concurrent one. However, this cannot necessarily be generalized to multiple memory locations; the sequential ordering of events for one location may conflict (i.e. create a cycle) with the sequential ordering for another location.

2.3 The full set of requirements

SC-Per-Location is one of the four requirements of this framework. It is the only requirement described solely in terms of executions; the other three are defined in terms of a particular architecture. This requires a formal definition of an architecture.

Definition. An *architecture* is a function $\mathscr{A}$ which maps executions $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ to tuples

$$(\mathsf{ppo}, \mathsf{fence}, \mathsf{prop})$$

such that for all executions E,

- $\mathsf{ppo} \subseteq \mathsf{po}$
- fence is some relation on events
- prop is some relation on the writes of $\mathbb{E}$ (not necessarily to the same location)

Here, the relation ppo ("preserved program order") refers to some subset of the program order that relates events which aren't allowed to be reordered in an execution, fence refers to pairs of events which are separated by a fence, and prop ("propagation order") refers to additional constraints (beyond those specified by co) on the order in which events get propagated to memory.

This definition formulates the notion of an architecture as a set of further restrictions on executions. Depending on how we define the orders ppo, fence, and prop on an execution, our model will satisfy different memory constraints, because our constraints are defined in terms of these relations. The set of all possible architectures that can be specified from this framework corresponds to all the different ways we can define these relations in terms of a given execution.

The full set of weak memory requirements is as follows. Let $\mathscr{A}$ be an architecture. Then for any execution $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$, we require

$$\begin{array}{rl} \text{(SC-Per-Location)} & \mathsf{acyclic}(\mathsf{pol} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}) \\ \text{(No Thin Air)} & \mathsf{acyclic}(\mathsf{hb}) \\ \text{(Observation)} & \mathsf{irreflexive}(\mathsf{fre}; \mathsf{prop}; \mathsf{hb}^*) \\ \text{(Propagation)} & \mathsf{acyclic}(\mathsf{co} \cup \mathsf{prop}) \end{array}$$

where

$$\mathsf{hb} = \mathsf{ppo} \cup \mathsf{fence} \cup \mathsf{rfe},$$

$$\mathsf{rfe} = \{(x,y) \mid x \xrightarrow{\mathsf{rf}} y \text{ and } \mathsf{proc}(x) \neq \mathsf{proc}(y)\},$$

and

$$\mathsf{fre} = \{(x,y) \mid x \xrightarrow{\mathsf{fr}} y \text{ and } \mathsf{proc}(x) \neq \mathsf{proc}(y)\},$$

and hb^* is the reflexive transitive closure of hb.

SC-Per-Location was described above; the other three requirements are discussed thoroughly in [2], and are best understood in the context of the various examples provided in that work. We present the full framework here for completeness, but our investigation into these properties was limited to SC-Per-Location.

2.4 SC-Per-Location: an alternate definition

The definition we have for SC-Per-Location makes intuitive sense - it corresponds directly to the classic definition of sequential consistency. However, as it turns out, this definition is equivalent to a seemingly weaker property (originally introduced in [1]), which we reproduce below.

Definition. An execution $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ satisfies the property *SC-Per-Location-2* if

$$\forall x,y \in \mathbb{E},\ x \xrightarrow{\mathsf{pol}} y \implies \neg(y \xrightarrow{\mathsf{com}^+} x)$$

i.e. no two events be related by pol in one direction and com^+ in the other direction.

This alternate definition captures the intuition that if an event precedes another event in program order, it cannot have a communication dependency (or a sequence of dependencies) on the latter event. Clearly, the existence of such a dependency would create a cycle in $\mathsf{pol} \cup \mathsf{com}$, and so it is easy to see that SC-Per-Location implies SC-Per-Location-2. As it turns out, this definition of SC-Per-Location-2 is actually equivalent to the one given in Section 2.2; this was first proved in Alglave [1] and we give a new proof of this result in the next section.

Now, as it turns out, the com^+ relation can be written as the union of the five relations $\mathsf{rf}, \mathsf{co}, \mathsf{fr}, \mathsf{co};\mathsf{rf}$, and $\mathsf{fr};\mathsf{rf}$. We state this as a theorem, and provide a sketch of the proof.

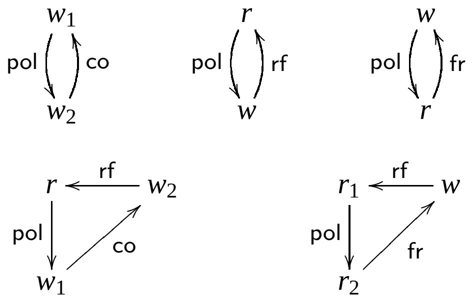


Figure 3: The five patterns prohibited by SC-Per-Location-2.

Theorem 2.1. *Let $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ be an execution. Then we have*

$$\mathsf{com}^+ = \mathsf{com} \cup (\mathsf{co};\mathsf{rf}) \cup (\mathsf{fr};\mathsf{rf}).$$

Proof. Suppose we have a path $x \to p_1 \to \cdots \to p_k \to y$, where $\to$ abbreviates $\xrightarrow{\mathsf{com}}$. We proceed by induction on k. If $k = 0$, we have $x \xrightarrow{\mathsf{com}} y$, and we are done.

Now, suppose $k \geq 1$ and assume inductively that the theorem holds for the all shorter paths. We have

$$x \to p_1 \to \cdots \to p_k \to y.$$

Now, the path $p_1 \to \cdots \to p_k \to y$ is a shorter path, and hence by our induction hypothesis, we have $p_1 \xrightarrow{\mathsf{com}} y$, $p_1 \xrightarrow{\mathsf{co;rf}} y$, or $p_1 \xrightarrow{\mathsf{fr;rf}} y$. Furthermore, we have $x \xrightarrow{\mathsf{co}} p_1$, $x \xrightarrow{\mathsf{rf}} p_1$, or $x \xrightarrow{\mathsf{fr}} p_1$. If we consider all these cases (many of which are vacuous due to the fact that co, rf and fr all relate events of specific types), it is easy to demonstrate that $x \xrightarrow{\mathsf{com}} y$, $x \xrightarrow{\mathsf{co;rf}} y$, or $x \xrightarrow{\mathsf{fr;rf}} y$. □

From this theorem, we can clearly see that an execution satisfies SC-Per-Location-2 if and only if it does not contain any of the patterns in Figure 3. We will ultimately prove that SC-Per-Location is equivalent to SC-Per-Location-2, which guarantees that a cycle of any kind in $\mathsf{pol} \cup \mathsf{com}$, no matter how big the cycle is, will imply the existence of a "mini"-cycle of one of these five variants.

2.5 An equivalence theorem

Before we state and prove the equivalence theorem (originally proved in [1], but proved here in a somewhat more straightforward manner), we first establish two simple lemmas.

Lemma 2.2. *The relation com^+ is irreflexive.*

Proof. Suppose $x \xrightarrow{\mathsf{com}^+} x$. By Theorem 2.1, we have three cases.

Case 1: $x \xrightarrow{\mathsf{com}} x$. This is impossible; co is irreflexive by definition (it is an irreflexive total order), and rf and fr are both trivially irreflexive because they only relate events of different types.

Case 2: $x \xrightarrow{\mathsf{co;rf}} x$. This is impossible; $\mathsf{co};\mathsf{rf}$ relates writes to reads, and hence is irreflexive.

Case 3: $x \xrightarrow{\mathsf{fr;rf}} x$. Then there exists an event z with $x \xrightarrow{\mathsf{fr}} z \xrightarrow{\mathsf{rf}} x$; this in turn implies the existence of an event y with $y \xrightarrow{\mathsf{rf}} x$, $y \xrightarrow{\mathsf{co}} z$, and $z \xrightarrow{\mathsf{rf}} x$. By the uniqueness of writes for the rf relation, we must have $y = z$; therefore $y \xrightarrow{\mathsf{co}} y$, which is impossible since co is irreflexive. □

Upon examination of Figure 2b, it is intuitively clear that any two events in this picture either on the same "level", or there is a path from one to the other. This is precisely what Lemma 2.3 says.

Lemma 2.3. *Let* $E = (\mathbb{E}, \mathsf{po}, \mathsf{co}, \mathsf{rf})$ *be an execution, and let* $x, y \in \mathbb{E}$ *with* $\mathsf{addr}(x) = \mathsf{addr}(y)$. *Then one of the following holds:*

1. $x \xrightarrow{\mathsf{com}^+} y$
2. *x and y are both writes, and* $x = y$
3. *x and y are both reads, and* $\mathsf{rf}^{-1}(x) = \mathsf{rf}^{-1}(y)$
4. $y \xrightarrow{\mathsf{com}^+} x$.

Proof. We have four cases, corresponding to x and y each being either reads or writes; however, the symmetry of the read-write cases reduces the number to three. In all three cases, the theorem reduces to the totality of co.

Case 1: x is a write, y is a write. Then by totality of co, either $x \xrightarrow{\mathsf{co}} y$, $y \xrightarrow{\mathsf{co}} x$, or $x = y$.

Case 2: x is a write, y is a read. Then by totality of co, either $x \xrightarrow{\mathsf{co}} \mathsf{rf}^{-1}(y)$, $x = \mathsf{rf}^{-1}(y)$, or $\mathsf{rf}^{-1}(y) \xrightarrow{\mathsf{co}} x$. In the first case, $x \xrightarrow{\mathsf{co;rf}} y$; in the second, $x \xrightarrow{\mathsf{rf}} y$; and in the third, $y \xrightarrow{\mathsf{fr}} x$.

Case 3: x is a read, y is a read. Then by totality of co, either $\mathsf{rf}^{-1}(x) \xrightarrow{\mathsf{co}} \mathsf{rf}^{-1}(y)$, $\mathsf{rf}^{-1}(x) = \mathsf{rf}^{-1}(y)$, or $\mathsf{rf}^{-1}(y) \xrightarrow{\mathsf{co}} \mathsf{rf}^{-1}(x)$. In the first case, $x \xrightarrow{\mathsf{fr;rf}} y$; in the second, we are done; and in the third, $y \xrightarrow{\mathsf{fr;rf}} x$. □

Theorem 2.4. *Let E be an execution. Then E satisfies SC-Per-Location if and only if E satisfies SC-Per-Location-2.*

Proof. It is clear that SC-Per-Location implies SC-Per-Location-2.

We prove the other direction by contrapositive. Suppose SC-Per-Location does not hold; that is, there exists a cycle in $\mathsf{pol} \cup \mathsf{com}$. Clearly any such cycle is also a cycle in $\mathsf{pol} \cup \mathsf{com}^+$ (since $\mathsf{com} \subseteq \mathsf{com}^+$). We proceed by induction on the length of this cycle, noting trivially that the length cannot be 1 (because we know that pol and com are both irreflexive).

If the cycle has length two, we must either have $x \xrightarrow{\mathsf{pol}} p \xrightarrow{\mathsf{com}^+} x$ or $x \xrightarrow{\mathsf{com}^+} p \xrightarrow{\mathsf{pol}} x$, because both of these relations are by themselves acyclic. In either case, the SC-Per-Location-2 condition is clearly violated by x and p.

Suppose the cycle has length three or more, i.e.

$$x \to p_1 \to p_2 \to \cdots \to x,$$

where $\to$ abbreviates the union of pol and com^+. Also, inductively assume that the existence of a shorter cycle implies that SC-Per-Location-2 does not hold. Assume that $x \xrightarrow{\mathsf{com}^+} p_1 \xrightarrow{\mathsf{pol}} p_2$ or $x \xrightarrow{\mathsf{pol}} p_1 \xrightarrow{\mathsf{com}^+} p_2$, because otherwise it is clear by transitivity of com^+ and pol that we can obtain a shorter cycle $x \to p_2 \to \cdots \to x$, and so by our inductive hypothesis SC-Per-Location-2 doesn't hold. Then we have several cases, based on Lemma 2.3.

Case 1: $x \xrightarrow{\mathsf{com}^+} p_2$. Then we have the shorter cycle $x \xrightarrow{\mathsf{com}^+} p_2 \to \cdots \to x$, and so by our inductive hypothesis, SC-Per-Location-2 does not hold.

Case 2: x and p_2 are writes where $x = p_2$. Then clearly $x = p_2 \to \cdots \to x$ is a shorter cycle, so by our inductive hypothesis, SC-Per-Location-2 does not hold.

Case 3a: x and p_2 are reads where $\mathsf{rf}^{-1}(x) = \mathsf{rf}^{-1}(p_2)$, and $x \xrightarrow{\mathsf{com}^+} p_1 \xrightarrow{\mathsf{pol}} p_2$. Then it is straightforward to show that $p_2 \xrightarrow{\mathsf{com}^+} p_1$, giving $p_1 \xrightarrow{\mathsf{pol}} p_2 \xrightarrow{\mathsf{com}^+} p_1$, which violates SC-Per-Location-2.

Case 3b: x and p_2 are reads where $\mathsf{rf}^{-1}(x) = \mathsf{rf}^{-1}(p_2)$, and $x \xrightarrow{\mathsf{pol}} p_1 \xrightarrow{\mathsf{com}^+} p_2$. Then it is straightforward to show that $p_1 \xrightarrow{\mathsf{com}^+} x$, giving $x \xrightarrow{\mathsf{pol}} p_1 \xrightarrow{\mathsf{com}^+} x$, which violates SC-Per-Location-2.

Case 4a: $p_2 \xrightarrow{\mathsf{com}^+} x$, and $x \xrightarrow{\mathsf{com}^+} p_1 \xrightarrow{\mathsf{pol}} p_2$. Then clearly $p_2 \xrightarrow{\mathsf{com}^+} p_1$, giving $p_1 \xrightarrow{\mathsf{pol}} p_2 \xrightarrow{\mathsf{com}^+} p_1$, which violates SC-Per-Location-2.

Case 4b: $p_2 \xrightarrow{\mathsf{com}^+} x$, and $x \xrightarrow{\mathsf{pol}} p_1 \xrightarrow{\mathsf{com}^+} p_2$. Then clearly $p_1 \xrightarrow{\mathsf{com}^+} x$, giving $x \xrightarrow{\mathsf{pol}} p_1 \xrightarrow{\mathsf{com}^+} x$, which violates SC-Per-Location-2.

By Lemma 2.3 there are no other possibilities. Therefore by induction, if SC-Per-Location does not hold then SC-Per-Location-2 does not hold, and the proof is complete. □

We believe this proof is new. Its direct use of an inductive argument and a "totality" lemma (Lemma 2.3) for com^+ both distinguishes it from the original [1], and makes its mechanization in ACL2 much easier. One of ACL2's big strengths is its ability to prove theorems inductively, and by understanding an inductive hand proof of this theorem, we were able to make the ACL2 proof much more straightforward.

3 ACL2 Mechanization

In this section we present our ACL2 mechanization of the framework and proofs presented above. We make extensive use of the `defun-sk` construct; our definitions of the relations po, co, rf, and fr, as well as various combinations of these relations, are introduced with `defun-sk` in order to make the concepts as general as possible; instead of defining them in terms of a specific data structure (like a graph), we define them as completely general relations which satisfy only the properties we require.

For clarity, we have chosen to present the ACL2 mechanization in a separate section from the preceding one. We have also opted to reproduce most of the definitions, theorems, and even a few key lemmas in order to give the reader a fuller understanding of how these ideas were mechanized. The interested reader might gain some insight into reading the ACL2 code carefully, but is encouraged to skim through it if necessary.

3.1 Mechanization of Concurrent Executions

We formalize the concepts of events, po, co, and rf as constrained functions that satisfy the requirements given in the previous section.

```
(encapsulate
 (((writep *) => *)
  ((readp *) => *)

  ((addr *) => *)
  ((proc *) => *)

  ((po * *) => *)
  ((rf * *) => *)
  ((co * *) => *)
```

```
    ((rf-inv-fn *) => *))

    ; ... constraints omitted
  )
```

The required properties of these functions are guaranteed by a number of exported theorems, such as totality of po on events in the same process, totality of co on writes to the same location, and the one-sided invertibility of rf (this last property implicitly make use of rf's inverse function rf-inv-fn).

We define the function fr in terms of co and rf using ACL2's defun-sk construct:

```
(defun-sk fr (x z)
  (exists y
    (and (rf y x) (co y z))))
```

We define the ACL2 analogues of sequenced relations co; rf and fr; rf similarly:

```
(defun-sk co->rf (x z)
  (exists y
    (and (co x y) (rf y z))))
(defun-sk fr->rf (x z)
  (exists y
    (and (fr x y) (rf y z))))
```

We define the functions com and pol as expected:

```
(defun com (x y)
  (or (co x y)
      (rf x y)
      (fr x y)))
(defun pol (x y)
  (and (po x y)
       (equal (addr x) (addr y))))
```

The transitive closure of com is defined in terms of the existence of a path:

```
(defun com-pathp (path x y)
  (cond ((endp path) (com x y))
        (t (and (com x (car path))
                (com-pathp (cdr path) (car path) y)))))
(defun-sk com+ (x y)
  (exists path (com-pathp path x y)))
```

The variable path represents the elements between (and not including) x and y. We prove that we can rewrite com+ according to Theorem 2.1:

```
(defthm rewrite-com+
  (equal (com+ x y)
         (or (com x y)
             (co->rf x y)
             (fr->rf x y))))
```

We prove that com+ is irreflexive, corresponding to Lemma 2.2:

```
(defthm com+-irreflexive
  (not (com+ x x)))
```

And we prove a theorem about the “totality” of com+, corresponding to Lemma 2.3:

```
(defthm com+-totality
  (implies (and (or (readp x) (writep x))
                (or (readp y) (writep y))
                (equal (addr x) (addr y))
                (not (com+ x y))
                (not (and (writep x)
                          (writep y)
                          (equal x y)))
                (not (and (readp x)
                          (readp y)
                          (equal (rf-inv-fn x) (rf-inv-fn y)))))
           (com+ y x)))
```

The majority of these theorems were proven by ACL2 with no hints other than the occasional instantiation of witness functions and the selective enabling/disabling of functions and theorems.

3.2 Mechanization of both definitions of SC-Per-Location

In order to define SC-Per-Location in ACL2, we need to define the notion of a “cycle” in the union of pol and com. We first define the union of these two relations:

```
(defun pol-com (x y)
  (or (pol x y)
      (com x y)))
```

Then we define the notion of a path in pol-com:

```
(defun pol-com-pathp (path x y)
  (cond ((endp path) (pol-com x y))
        (t (and (pol-com x (car path))
                (pol-com-pathp (cdr path) (car path) y)))))
```

If path is nil, this definition reduces to (pol-com x y). Now, we can define a cycle in pol-com as

```
(defun pol-com-cyclep (cycle x)
  (pol-com-pathp cycle x x))
```

SC-Per-Location states that there does not exist a cycle in pol-com. This can be stated as

$$(\forall \texttt{x}, \texttt{cycle})\ \texttt{(not (pol-com-cyclep cycle x))}.$$

We can thus define SC-Per-Location in ACL2 as

```
(defun-sk sc-per-location-1 ()
  (forall (x cycle)
          (not (pol-com-cyclep cycle x))))
```

SC-Per-Location-2 can be easily defined as

```
(defun-sk sc-per-location-2 ()
  (forall (x y)
          (implies (pol x y)
                   (not (com+ y x)))))
```

3.3 Mechanization of the equivalence proof, Part 1

As before, the easy part of the equivalence proof is the fact that `(sc-per-location-1)` implies `(sc-per-location-2)`. The first step involved proving an unquantified version of the theorem, where we assume `(pol x y)` and `(com+ y x)`, and consider the three cases afforded by `rewrite-com+`:

```
(defthm pol-com-cycle
  (implies (and (pol x y)
                (com y x))
           (pol-com-cyclep (list y) x)))
(defthm pol-co->rf-cycle
  (implies (and (pol x y)
                (co->rf y x))
           (pol-com-cyclep (list y (co->rf-witness y x)) x)))
(defthm pol-fr->rf-cycle
  (implies (and (pol x y)
                (fr->rf y x))
           (pol-com-cyclep (list y (fr->rf-witness y x)) x)))
```

Then we add `sc-per-location-1` back into these theorems with `:instance` hints:

```
(defthm pol-com-not-sc-per-location-1
  (implies (and (sc-per-location-1)
                (pol x y))
           (not (com y x)))
  :hints (("Goal"
           :use ((:instance sc-per-location-1-necc
                            (x x)
                            (potential-cycle (list y)))))))
(defthm pol-co->rf-not-sc-per-location-1
  (implies (and (sc-per-location-1)
                (pol x y))
           (not (co->rf y x)))
  :hints (("Goal"
           :use ((:instance sc-per-location-1-necc
                  (x x)
                  (potential-cycle (list y (co->rf-witness y x))))))))
(defthm pol-fr->rf-not-sc-per-location-1
  (implies (and (sc-per-location-1)
                (pol x y))
           (not (fr->rf y x)))
  :hints (("Goal"
           :use ((:instance sc-per-location-1-necc
                  (x x)
                  (potential-cycle (list y (fr->rf-witness y x))))))))
```

Finally, we state the fully quantified version of the theorem, which ACL2 proves immediately:

```
(defthm sc-per-location-1-implies-2
  (implies (sc-per-location-1)
           (sc-per-location-2)))
```

3.4 Mechanization of the equivalence proof, Part 2

The proof that `(sc-per-location-2)` implies `(sc-per-location-1)` was broken down into 4 steps:

1. Prove that any 2-cycle in `pol-com+` violates `sc-per-location-2`, and that if there is a cycle of length 3 or greater in `pol-com+`, where `pol-com+` is the union of `pol` and `com+`, then there is a smaller cycle in `pol-com+`, and
2. Use the theorem in step 1 to define a function, `collapse-cycle`, which takes a cycle in `pol-com+` and produces a pair `(x y)` such that `(pol x y)` and `(com+ y x)`
3. Combine steps 1 and 2 to show that if we have a cycle in `pol-com` (i.e. a violation of `sc-per-location-1`), we have a pair `(x y)` which violates `sc-per-location-2`

Step 1 is summarized by two theorems, one that states that 2-cycles in `pol-com+` violate `sc-per-location-2`, and one that takes cycles longer than 2 and produces a smaller cycle.

```
(defthm cycle-2
  (implies (and (pol-com+-cyclep cycle x)
                (endp (cdr cycle))
                (not (and (pol x (car cycle))
                          (com+ (car cycle) x))))
           (and (pol (car cycle) x)
                (com+ x (car cycle)))))
(defthm collapse-cycle-thm
  (implies (and (not (pol-com+-cyclep (list p1) x))
                (not (pol-com+-cyclep (list* p2 rst) x))
                (not (pol-com+-cyclep rst x))
                (not (pol-com+-cyclep (list p2) p1)))
           (not (pol-com+-cyclep (list* p1 p2 rst) x))
  :hints (("Goal"
           :cases ((com+ x p2)
                   (and (writep x)
                        (writep p2)
                        (equal x p2))
                   (and (readp x)
                        (readp p2)
                        (equal (rf-inv-fn x) (rf-inv-fn p2)))
                   (com+ p2 x)))))
```

Notice that the case split corresponds exactly to Theorem 2.3, just as in the written proof.

For Step 2, we define the function `collapse-cycle` to shorten the cycle according to the previous theorem. The `collapse-cycle` function satisfies the property that if it is given a violation of `sc-per-location-1`, it produces a violation of `sc-per-location-2`:

```
(defun collapse-cycle (cycle x)
  (let* ((p1 (car cycle))
         (p2 (cadr cycle))
         (rst (cddr cycle)))
    (cond ((endp cycle) (mv nil x))
          ((endp (cdr cycle))
           (if (pol x (car cycle))
               (mv x (car cycle))
             (mv (car cycle) x)))
          ((pol-com+-cyclep (list* p2 rst) x)
           (collapse-cycle (list* p2 rst) x))
          ((pol-com+-cyclep rst x)
           (collapse-cycle rst x))
          ((pol-com+-cyclep (list p2) p1)
           (collapse-cycle (list p2) p1))
          (t (collapse-cycle (list p1) x)))))
(defthm collapse-cycle-pol-com+
  (implies (pol-com+-cyclep cycle x)
           (mv-let (new-x new-y)
                   (collapse-cycle cycle x)
                   (and (pol new-x new-y)
                        (com+ new-y new-x)))))
```

For Step 3, we first add in the quantifier for `sc-per-location-2`:

```
(defthm sc-per-location-1-implies-2-unquantified
  (implies (sc-per-location-2)
           (not (pol-com-cyclep cycle a)))
  :hints (("Goal"
           :use ((:instance sc-per-location-2-necc
                            (x (mv-let (new-x new-y)
                                       (collapse-cycle cycle a)
                                       (declare (ignore new-y))
                                       new-x))
                            (y (mv-let (new-x new-y)
                                       (collapse-cycle cycle a)
                                       (declare (ignore new-x))
                                       new-y)))))))
```

The result follows immediately:

```
(defthm sc-per-location-2-implies-1
  (implies (sc-per-location-2)
           (sc-per-location-1)))
```

3.5 Mechanizing the other requirements

The other requirements of this framework were also mechanized in ACL2, using constrained functions to represent ppo, fence, and prop, and with rfe, fre, and hb defined in terms of these constrained functions.

The concepts of No Thin Air, Observation, and Propagation were defined as follows:

```
(defun-sk no-thin-air ()
  (forall (x potential-cycle)
          (not (hb-cyclep potential-cycle x))))
(defun-sk observation ()
  (forall x
          (not (fre->prop->hb* x x))))
(defun-sk propagation ()
  (forall (x cycle)
          (not (co-prop-cyclep cycle x))))
```

We did not investigate these requirements to the extent that we analyzed SC-Per-Location. We reproduce their definitions here for completeness.

4 Conclusions

In this work, we presented an ACL2 mechanization of a generic framework for weak memory, as well as a novel proof of an established result for this framework. We hope to incorporate this framework into our ongoing research into how a theorem prover like ACL2 can be used to verify correctness properties of real-world concurrent programs. Our most immediate future work consists of applying these concepts (actually, a simplification of these concepts) to proofs on a multi-processor x86 model, but this work suggests the possibility of applying a general weak memory framework to other models as well.

This work was supported by NSF. We gratefully acknowledge the many helpful comments and discussions provided by Jade Alglave and Matt Kaufmann.

References

[1] Jade Alglave (2010): *A Shared Memory Poetics*. Ph.D. thesis, Université Paris 7.

[2] Jade Alglave, Luc Maranget & Michael Tautschnig (2014): *Herding Cats - Modelling, simulation, testing, and data-mining for weak memory*. *TOPLAS (to appear)*. Available at http://arxiv.org/abs/1308.6810.

[3] Gérard Boudol & Gustavo Petri (2009): *Relaxed Memory Models: An Operational Approach*. *SIGPLAN Not.* 44(1), pp. 392–403, doi:10.1145/1594834.1480930.

[4] Nathan Chong & Samin Ishtiaq (2008): *Reasoning About the ARM Weakly Consistent Memory Model*. In: *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, MSPC '08, ACM, New York, NY, USA, pp. 16–19, doi:10.1145/1353522.1353528.

[5] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *Commun. ACM* 12(10), pp. 576–580, doi:10.1145/363235.363259.

[6] L. Lamport (1979): *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. *IEEE Trans. Comput.* 28(9), pp. 690–691, doi:10.1109/TC.1979.1675439.

[7] Scott Owens, Susmit Sarkar & Peter Sewell (2009): *A Better x86 Memory Model: X86-TSO*. In: *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, Springer-Verlag, Berlin, Heidelberg, pp. 391–407, doi:10.1007/978-3-642-03359-9_27.

[8] Susan Owicki & David Gries (1976): *An Axiomatic Proof Technique for Parallel Programs*. *Acta Informatica* 6, pp. 319–340, doi:10.1007/BF00268134.

[9] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget & Derek Williams (2011): *Understanding POWER Multiprocessors*. *SIGPLAN Not.* 46(6), pp. 175–186, doi:10.1145/1993316.1993520.

[10] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen & Jade Alglave (2009): *The Semantics of x86-CC Multiprocessor Machine Code*. *SIGPLAN Not.* 44(1), pp. 379–391, doi:10.1145/1594834.1480929.