Computer Science Technical Report

The Location Consistency Memory Model and Cache Protocol: Specification and Verification

Charles Wallace, Guy Tremblay and José Nelson Amaral

Michigan Technological University
Computer Science Technical Report
CS-TR-01-01

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

The Location Consistency Memory Model and Cache Protocol: Specification and Verification

Charles Wallace
Computer Science Dept.
Michigan Technological University
Houghton, Michigan, USA

Guy Tremblay
Dépt. d'informatique
Université du Québec à Montréal
Montréal, Québec, Canada

José Nelson Amaral
Computing Science Dept.
University of Alberta
Edmonton, Alberta, Canada

Abstract

We use the *Abstract State Machine* methodology to give formal operational semantics for the *Location Consistency* memory model and cache protocol. With these formal models, we prove that the cache protocol satisfies the memory model, but in a way that is strictly stronger than necessary, disallowing certain behavior allowed by the memory model.

1 Introduction

A *shared memory multiprocessor machine* is characterized by a collection of processors that exchange information with one another through a *global address space* [1, 6]. In such a machine, processors access memory locations concurrently through standard *read (load)* and *write (store)* memory instructions. Shared memory machines have many buffers where data written by a processor can be stored before it is shared with other processors. As a consequence, multiple values written to a single memory location may coexist in the system. For instance, the local caches of various processors might contain different values written to the same memory location.[1]

The programs running on a shared memory machine are profoundly affected by the order in which memory operations are made visible to processors. For instance, when a processor performs a read operation, the value it reads depends on which previous write operations are currently visible. A *memory consistency model* is a contract between a program and the underlying machine architecture that constrains the order in which memory operations appear to be performed with respect to one another (*i.e.*, become visible to processors) [6]. By constraining the order of operations, a memory consistency model determines which values can legally be returned by each read operation. The implementation of a memory consistency model in a shared memory machine with caches requires a *cache protocol*, which invalidates or updates cached values when they no longer represent readable values according to the memory model.

[1] There may be multiple values per location even in a uniprocessor system, if it allows writers to bypass one another in write buffers or in the cache.

The most common memory consistency model, *sequential consistency* (SC) [17], ensures that memory operations performed by the various processors are *serialized* (*i.e.*, seen in the same order by all processors). This results in a model similar to the familiar uniprocessor model. A simple way to implement SC on a shared memory multiprocessor is to define a notion of *ownership* of a memory location and require a processor to become the owner of a location before it can update its value. The serialization of memory operations is obtained by restricting ownership of a location to one processor at a time.

In the SC model, at any time there is a unique most recent write to a location, and any other values stored in the system for that location are not legally readable; they must either be invalidated or updated. Thus a major drawback of SC is the high level of interprocessor communication required by the cache protocol. Each new write may lead to the invalidation or update of many values cached in other processors. Because of the requirement that all write memory operations be serialized, the SC model is quite restrictive and is thus said to be a *strong* memory model. *Weaker* memory models have been proposed to relax the requirements imposed by SC. Examples include *release consistency* [11], *lazy release consistency* [15], *entry consistency* [2], and *DAG consistency* [3]. Relaxed memory models place fewer constraints on the memory system than SC. Eliminating constraints permits more parallelism and requires less interprocessor communication but complicates reasoning about program behavior.

All these models have the *coherence* property. In a coherent memory model, all writes become visible to other processors, and all the writes in the system are seen in the same order by all processors. In 1994, Gao and Sarkar proposed the *Location Consistency (LC) memory model* [9], one of the weakest memory models proposed to date. LC is the only model that does not ensure coherence. Under LC, memory operations performed by multiple processors need not be seen in the same order by all processors. Instead, the content of a memory location is seen as a partially ordered set of values. Multiple values written to the same memory location can be stored in write buffers, local caches, and in the main memory. Because LC allows the coexistence of multiple legal values to the same location, there is no need to invalidate or update remote cached values, as is the case with SC. Hence the LC model has the potential to reduce the consistency-related traffic in the network significantly.

In their more recent paper [10], Gao and Sarkar describe both the LC memory model and a cache protocol, the *LC cache protocol*, that implements the LC model. They describe the LC model in terms of an "abstract interpreter". This interpreter maintains the state of each memory location as a partially ordered set of values. The partial order defines the set of legal values for a read operation according to the LC model. The LC cache protocol is designed for a machine in which each memory location has a single value stored in *main memory* and each processor may cache values for many memory locations.

An important requirement of a cache protocol is that the resulting machine is restricted to behavior allowed by the corresponding memory model. Gao and Sarkar present a proof that the cache protocol satisfies the memory model. However, their description of the memory model is based on an *ad hoc* operational semantics notation that is not rigorously defined. The description of the cache protocol is entirely informal and leaves some important assumptions unstated. A formal approach would bring clarity and precision to the memory model and cache protocol definitions, and it would provide a more convincing basis for a correctness proof.

In this paper, we specify the LC memory model and the LC cache protocol using the formal operational semantics methodology of *Abstract State Machines (ASM)* [12]. We use the original

descriptions by Gao and Sarkar as the basis for our models, augmented with a couple of assumptions that are implicit in the original descriptions. We then prove that the LC cache protocol correctly implements the LC memory model, *i.e.*, for a machine that implements the cache protocol, every read operation returns a legal value according to the memory model. In addition, we show that the LC cache protocol is *strictly stronger* than the LC memory model; *i.e.*, there are legal values according to the memory model that cannot be returned under the cache protocol.

An Abstract State Machine, like a Finite State Automaton (FSA) [20], a Turing Machine [21], or a Random Access Machine [5], is a mechanism that takes input and performs actions that lead to state transformations. These computing models have different notions of state, state transformation, and program (the set of rules that describe the possible state transformations). An ASM state is an *algebra* (a collection of function names interpreted over a given set), and a state transformation is a set of changes to the interpretations of these functions. A brief introduction to ASM is presented in Appendix A.

An interesting feature of ASM is that the specification of a model is not confined to a fixed level of representation, as is the case with other computing models, *e.g.*, the infinite tape of a Turing Machine or the infinite register sequence of a Random Access Machine. Moreover, both atomic state transformations (*e.g.*, a change of a single function interpretation at a single location) and complex transformations (*e.g.*, a change of multiple functions at different locations) can be interpreted as single abstract machine steps. These properties allow the unification of the models of the abstract interpreters for the memory model and the cache protocol. Furthermore, the operational nature of the ASM model is closer to Gao and Sarkar's description of the model than a purely declarative specification (*e.g.*, using a process algebra [18]) would be.

Our specifications of the LC memory model and of the LC cache protocol are similar in that they refine a common (top-level) specification. In §2 we define the common portions of the two models. The result is an ASM model LC_0. In §3 we refine this model to arrive at a model LC_{mm} of the LC memory model. In §4 we make different refinements, resulting in a model LC_{cp} of the LC cache protocol. In §5 we prove that LC_{cp} is an implementation of LC_{mm}, in the sense that every value read in a run of LC_{cp} is also read in an equivalent run of LC_{mm}. In §6 we prove that LC_{cp} is strictly stronger than LC_{mm}, in the sense that there are readable values in a run of LC_{mm} that are not read in any equivalent run of LC_{cp}. We conclude in §7 with some directions for future work.

2 Shared memory system and memory operations

In a shared memory machine, processors can reference a shared set of memory locations, organized in a global shared address space. The operations that processors use to access these shared locations are no different from those they use to access their local memory locations. Although it may appear intuitive to think that each shared memory location holds a single value at any given time, cache memories provide multiple places to store values for a single location. Thus at any given time, multiple values may be simultaneously associated with the same memory location.

A processor can perform four types of operation on a memory location.

- A *read* retrieves a value associated with a location, possibly storing it in some area local to the processor.
- A *write* adds a value to the set of values associated with the location.

In real systems, the number of places available to store the set of values associated with a location is finite. Therefore, a side effect of a read or write operation is that a value previously associated with a given location may no longer be available in the system.

- An *acquire* grants exclusive ownership of a location to a processor.[2]

  The exclusive ownership of a location imposes a sequential order on processor operations. Hence when it is useful to have a unique global "most recent write" to a location, such write can be defined as the most recent write by a processor that owned the location at the time of the write. When acquiring a location, a processor updates its own state by discarding any old value it has stored for the location.

- A *release* operation takes exclusive ownership away from a processor. A processor that owns a location must release it before any other processor can acquire it. Any processor attempting to acquire a location currently owned by another processor must wait until the location is released by its current owner. If the releasing processor has written to the location, the release operation has the additional effect of making the value of its most recent write available to other processors. In this way, a processor that subsequently acquires the location will have access to the value of the global "most recent write".

Gao and Sarkar do not speak of acquire and release operations separately; rather, they speak of acquire-release *pairs* of operations. Thus it is assumed that a processor must gain ownership of a location through an acquire before releasing that location. Furthermore, if a processor acquires a location, it must later relinquish ownership through a release before any processor can perform another acquire on that location.

The model of the LC memory model (LC_{mm}) in §3 and the model of the LC cache protocol (LC_{cp}) in §4 both require formalizations of the notions discussed above. In the rest of this section, we define a higher-level ASM model LC_0 to represent these notions in a generic way. LC_0 models only the the general control flow associated with the execution of the memory operations, including the waiting associated with an acquire operation. In this initial model, the flow of data is ignored. Later, we refine LC_0 to the models LC_{mm} and LC_{cp}, adding details appropriate to each of these models (partial order of operations vs. cache information).

LC_0: Universes and agents

In this section, we present the universes used in all our ASM models. We assume that the multiprocessor system has a fixed set of processors, a fixed set of memory locations, and a fixed set of data values. These sets are represented in LC_0 as the Processor, Location and Value universes, respectively. We also define an OpType universe, comprising the four types of operation: read, write, acquire and release.

A distributed computation in ASM is modeled as a multi-agent computation in which agents execute concurrently and where each agent performs a sequence of state transformations. To define the agents of the model, we consider the various forms of concurrency that can exist between

[2]Exclusive ownership is used in SC to restrict the processors reading or writing to a single location to one at a time. In SC, only a processor that owns a location may perform a read or write on it. However, this is not the case with all memory models. As we shall see, LC has a notion of exclusive ownership, but it also allows processors without ownership to perform reads and writes. In LC, exclusive ownership does not mean exclusive read/write access.

Function	Profile/Description
p.loc	ProcAgent → Location Memory location on which agent p performs operations (static).
p.proc	ProcAgent → Processor Processor on behalf of which agent p performs operations (static).
p.opType	ProcAgent → OpType Type of operation that agent p performs in the *current* step.
p.nextOpType	ProcAgent → OpType Type of operation agent p will perform at the *next* step (monitored).
p.waiting?	ProcAgent → Boolean Is p waiting to acquire ownership of its location?
p.writeVal	ProcAgent → Value Value to be written in the current write operation by agent p (monitored).

Figure 1: Attributes of ProcAgents.

actions. We must then distinguish those actions that *should not* be performed concurrently from those that *may* but *need not* be performed concurrently, and from those that *must* be performed concurrently. To ensure that two actions are performed concurrently, a single agent can be defined to perform these actions using a "block rule"; to allow (but not require) two actions to be performed concurrently, the actions can be assigned to different agents; finally, to preclude two actions from being done concurrently, they can be assigned to mutually exclusive clauses within a single agent.

In modern multiprocessors, a single processor may perform operations on different locations concurrently. Moreover, when both a relaxed memory model and write buffers are used, multiple outstanding writes to the same location may coexist in the system. Moreover, multiple processors may perform concurrent memory operations, either on the same location or on different locations. However, we assume that a given processor does not perform multiple concurrent memory operations on a given location. Therefore, in our abstract model, for each processor P and for each location ℓ there is a unique *agent* whose task is to perform operations on location ℓ *on behalf of* processor P. We call such agents *processor agents*, and we define a universe ProcAgent in LC_0 accordingly. In order to complete the definition of LC_0, we introduce two more universes of agents: InitAgent (initializer agent) and OwnAgent (ownership agent). We now describe each of these agents.

LC_0: Processor agents

A processor agent is characterized by the attributes loc and proc: loc is the location on which the agent performs actions, and proc is the processor on behalf of which the agent acts. Note that there may be multiple agents with the same proc attribute or the same loc attribute; indeed, such commonality exists in all but the degenerate case of a uniprocessor machine. Both attributes have fixed values, *i.e.*, they are never changed by the state transformations of the abstract machine. Thus in ASM terminology, they are called *static functions*.

Associated with each ProcAgent are some attributes whose values may change during an execution of the abstract machine; in ASM terminology, these are *dynamic functions*. For instance, the

Function	Profile/Description
i.loc	InitAgent → Location Memory location that agent i must initialize (static).
i.opType	InitAgent → OpType Type of operation that agent i performs in the current step.
i.writeVal	InitAgent → Value Value used to initialize the location of agent i (monitored).
ℓ.initialized?	Location → Boolean Has ℓ been initialized by its InitAgent?

Figure 2: Attributes of InitAgents and Locations.

attribute opType indicates the type of operation that the agent is to perform in the current step. Some operations may take multiple steps; for instance, a processor agent performing an acquire operation may need to wait for another processor agent to release ownership of its location. When the current operation is completed, the processor agent updates its opType attribute.

The type of the next operation that the agent is to perform (once all actions associated with the current operation have been completed) is given by the attribute nextOpType. Since we are not interested in how the sequence of operations performed by each processor agent is chosen, nextOpType is not specified in our formal model; we simply leave it to the environment to update this function. In ASM terminology [14], nextOpType is an *external* or *monitored* function. In contrast, the attribute opType is explicitly updated by agents (and never by the environment), so it is called a *dynamic internal* function or *controlled* function.[3]

The attribute waiting? (a controlled function) determines whether a processor agent is waiting for ownership of its location (as the result of an acquire operation). If a processor agent is unable to gain ownership immediately, it updates its waiting? attribute to true. Finally, the monitored function writeVal, associated with the ProcAgent universe, provides the value written by a write operation. This function is not used in LC_0 but is used in both LC_{mm} and LC_{cp}.

In Figure 1, we present the attributes for the processor agents with their *profile* and brief descriptions of each attribute. The profile of a function specifies the function's signature (types of arguments and result); for example, the application of the function loc to a ProcAgent produces a Location. Note that we use postfix notation for terms involving unary functions.

LC_0: Initializer agents

A question arises regarding the initial status of each location. If a processor agent reads from a location that has never been written to, it is not clear what the result should be. We avoid this complication by ensuring that each location is initialized with a proper value before the processor agents start to perform operations on it. To this end, we define the InitAgent ("initializer agent") universe. Each initializer agent performs operations on a specific location; thus each InitAgent has the loc attribute (a static function) associated with it. An initializer agent performs two operations

[3]In fact, the terms "monitored" and "controlled" are taken from Parnas [19].

Function	Profile/Description
o.loc	OwnAgent → Location Memory location whose ownership agent o controls (static).
ℓ.owner	Location → ProcAgent Agent that has acquired location ℓ and has not released it.
ℓ.nextOwner	Location → ProcAgent Agent that is to become the owner of location ℓ in the next request for ownership (monitored).

Figure 3: Attributes of OwnAgents and Locations.

to initialize its location, after which it terminates. The operations are:

- a write, that associates some value with the location, followed by
- a release. A release performed by an initializer agent is similar to but simpler than a release performed by a processor agent. Releases by initializer agents, unlike those by processor agents, do not affect ownership of locations, since initializer agents do not perform acquires and therefore never have ownership. However, a release by an initializer agent does have the effect of making the initial value available to all processor agents.

Like a processor agent, each initializer agent has an opType attribute and a writeVal attribute. Each initializer agent consults its writeVal attribute to obtain the initial value to write to its location. Since it performs a fixed sequence of operations, an initializer agent has no nextOpType attribute. The attributes associated with initializer agents are presented in Figure 2.

We associate the attribute initialized? with each location. When an initializer agent has initialized its location, it updates the initialized? attribute of the location to true, indicating that processor agents are now free to perform operations on the location. This attribute is shown in Figure 2.

LC_0: Ownership agents

A processor agent seeking ownership of a location can gain ownership as long as no other processor agent currently owns the location. If another agent does own the location, the agent wishing to acquire must wait to gain ownership at a later time. At any time, there may be multiple processor agents waiting for ownership of the same location. The decision as to which agent is granted ownership is beyond the control of any processor agent. Furthermore, the ownership arbitration policy is implementation-dependent and beyond the scope of our specification. Therefore we define the OwnAgent universe, whose members have the responsibility of arbitrating ownership of locations. Since the ownership of different locations can be granted concurrently, we associate a unique OwnAgent with each memory location.

Since each ownership agent controls ownership of a single location, each OwnAgent has a loc attribute (a static function). Each location has an owner attribute, indicating which processor agent (if any) has ownership of the location. When a processor agent releases a location, there may be other processor agents waiting to gain ownership. The monitored (oracle) function nextOwner

is an attribute associated with locations; it provides the next processor agent selected to receive ownership of the location. This monitored function is consulted by the OwnAgent. All these attributes are presented in Figure 3.

Terminology

We introduce the following terminology for agents and actions in runs of either LC_{mm} or LC_{cp}.

Definition *If a* ProcAgent *or* InitAgent p *makes a move Rd at which* p.opType = read*, we say that* p performs a read *(or simply* reads*) at Rd. (Similarly for write.)*

Definition *If a* ProcAgent *or* InitAgent p *makes a move A at which* p.opType = acquire *and* p.loc.owner $= p$*, we say that* p performs an acquire *(or simply* acquires*) at A. (Similarly for release.)*

LC_0: Conditions on runs

Some aspects of our models LC_0, LC_{mm} and LC_{cp} are outside the control of the ASM transition rules. First, our static functions must have certain properties. We restrict attention to runs in which the following conditions are true of the static functions loc and proc, in the initial state (and therefore every subsequent state) of any run:

Static condition 1 *For every* Processor P *and for every* Location ℓ*, there is a unique* ProcAgent p *for which* p.proc $= P$ *and* p.loc $= \ell$.
(The ProcAgent performs operations on ℓ, on behalf of P.)

Static condition 2 *For every* Location ℓ*, there is a unique* InitAgent i *for which* i.loc $= \ell$.
(The InitAgent performs operations on ℓ.)

Static condition 3 *For every* Location ℓ*, there is a unique* OwnAgent o *for which* o.loc $= \ell$.
(The OwnAgent performs operations on ℓ.)

Second, there are certain conditions that must be true of the dynamic functions in the initial state of any run. In both specifications, we restrict attention to runs in which the following conditions are true in the initial state:

Init condition 1 *For every* InitAgent i, i.opType = write.
(The first operation performed by an InitAgent is a write.)

Init condition 2 *For every* Location ℓ, ℓ.owner.undef? and not ℓ.initialized?.[4]
(Initially, all locations are uninitialized, and no ProcAgent has ownership of any location.)

Init condition 3 *For every* ProcAgent p, not p.waiting?.
(Initially, no ProcAgent is waiting for ownership.)

[4] In general, the notation t.undef?, where t is a term, means t is undefined, that is, t evaluates to the special value undef. In other words, t.undef? if and only if $t =$ undef. Similarly, t.def? is a shorthand notation to indicate that t is defined; that is, $t \neq$ undef.

Also, the monitored function nextOwner must produce "reasonable" values at every move of any run. The nextOwner attribute identifies the next processor agent to own the given location. Only a processor agent currently waiting to obtain ownership on the location should be granted ownership. Thus we restrict attention to runs in which the following condition is met at every move:

Run condition 1 *For every* Location ℓ*, if* ℓ.nextOwner.def?*, then* ℓ.nextOwner.loc $= \ell$ *and* ℓ.nextOwner.waiting?*.*

Finally, in order to remain faithful to Gao and Sarkar's description, we restrict our attention to runs in which acquires and releases come in matching pairs. We formalize this notion in the following two conditions:

Run condition 2 *If a* ProcAgent p *acquires at a move* A_p *and releases after* A_p*, then there is a move* R_p *after* A_p *at which* p *releases such that* p *does not acquire in* (A_p, R_p)*.*[5]

Run condition 3 *If a* ProcAgent p *releases at a move* R_p*, then there is a move* A_p *before* R_p *at which* p *acquires such that* p *does not release in* (A_p, R_p)*.*[6]

LC_0: Processor agent module

The behavior of a processor agent is presented as an ASM *module* (*i.e.*, a transition rule) in Figure 4. ASM uses Self to refer to the entity executing the module. In a distributed ASM model such as the one we are presenting, there are a number of universes of agents, each universe associated with a specific module.[7]

The structure of the ProcAgent module has a recurring pattern: based on the current opType, the actions specified by an appropriate abstract rule (*Read*, *Write*, *Acquire*, or *Release*) are performed. Note that these rules are redefined in §3, giving us a complete ASM model LC_{mm}. A different set of definitions for these same rules then appears in §4, resulting in a distinct ASM model LC_{cp}.

A ProcAgent may begin performing operations on its location once the location is initialized. While an operation is executed, the operation to be performed in the next step is obtained through the rule *Get next operation*. This rule simply consults the environment to determine what should be done in the next step. Note that a processor agent may update its opType attribute to undef. In this case, it temporarily stops performing operations but continues to execute its program, firing the rule *Get next operation*. Once the opType attribute has a "well-defined" (non-undef) value, the agent resumes operations.

The acquire case is slightly different because a processor agent must first acquire ownership of the location. Thus if the location is owned by another processor agent, the acquiring agent must wait. In ASM terms, execution of the ProcAgent module with opType = acquire does not change opType until the location has finally been acquired (i.e., Self.loc.owner = Self). As for the

[5] Note that we do not require p to release after an acquire. A processor agent may maintain ownership of a location indefinitely.

[6] An effect of this run condition is the exclusion of any release "unpaired" with a preceding acquire (and therefore done by a processor agent that is not the current owner of the location). A similar effect can be achieved without additional run conditions by defining the ProcAgent module in Figure 4 so that such releases are simply ignored. Gao and Sarkar make a stronger claim, however: unpaired releases simply do not occur. Hence we reject this alternative solution since it is less faithful to the original description of the LC model.

[7] In object-oriented programming terminology, an agent *universe* could be interpreted as a *class*, the *agents* in the universe as the *instances* of that class, and the universe's *module* as the *behavior* associated with the class.

```
module ProcAgent:
if Self.loc.initialized? then
    case Self.opType of
    read:    Read
    write:   Write
    acquire: Acquire
    release: Release
    undef:   Get next operation

rule Read:
Get next operation

rule Write:
Get next operation

rule Acquire:
if Self.loc.owner ≠ Self then Self.waiting? := true
else Get next operation

rule Release:
Self.loc.owner := undef
Get next operation

rule Get next operation:
Self.opType := Self.nextOpType
```

Figure 4: Module for processor agents (ProcAgent).

```
module InitAgent:
case Self.opType of
write:   InitWrite
release: InitRelease

rule InitWrite:
Self.opType := release

rule InitRelease:
Self.opType := undef
Self.loc.initialized? := true
```

Figure 5: Module for initializer agents (InitAgent).

release case, it is Run Condition 3 that ensures that the releasing agent indeed has ownership of the location, which entails that it is correct to release ownership (*i.e.*, update Self.loc.owner to undef).

LC_0: Initializer agent module

Figure 5 contains the module for an initializer agent. Initial Condition 1 stipulates that the opType attribute of each initializer agent has the value write. Therefore, the following sequence of actions is performed by each initializer agent:

- Perform the actions described by rule *InitWrite*: change the value of opType to release.
- Perform the actions described by rule *InitRelease*: change the value of opType to undef, and indicate that the location associated with the agent is now initialized.

After this sequence of actions, undef matches neither write nor release. Therefore, no further actions are performed by the initializer agent. Once the execution of an initializer agent is complete, the associated location is properly initialized, thereby enabling processor agents to perform operations on the location.

The execution of the initializer agent associated with a location ℓ ensures that the location is properly initialized and thus enables the processor agents associated with the location.

LC_0: Ownership agent module

Figure 6 contains the module for ownership agents. If the location associated with the ownership agent currently has no owner (according to the owner attribute), and the nextOwner attribute has a defined value, then according to Run Condition 1, the processor agent indicated by nextOwner is currently waiting to gain ownership of the location (*i.e.*, the waiting? attribute evaluates to true for that processor agent). Therefore the ownership agent grants ownership to the processor agent, updating its waiting? status to false and making it the owner. Note that the waiting? attribute's only role is to allow this interaction with the OwnAgent: once a ProcAgent updates its waiting? attribute to true, only the appropriate OwnAgent can update it to false when ownership is granted.

```
module OwnAgent:
if Self.loc.owner.undef? and Self.loc.nextOwner.def? then
    Self.loc.nextOwner.waiting? := false
    Self.loc.owner := Self.loc.nextOwner
```

Figure 6: Module for ownership agents (OwnAgent).

The same is true of the owner attribute: once an OwnAgent updates it to a particular ProcAgent p, only the ProcAgent p can change it (to undef, when it is releasing the location).

3 The LC memory model

In the previous section, we described a generic framework — in terms of abstract *Read*, *Write*, *Acquire* and *Release* rules — that provides the top-level description of both the memory consistency model (LC_{mm}) and the cache protocol model (LC_{cp}). In this section we present a complete specification for the LC model, LC_{mm}, by defining the transition rules according to the memory model specifications.

The state of a memory system is determined entirely by the operations that have been performed upon the system. Following Gao and Sarkar [10], we view the state of a memory system as a *history* (*i.e.*, partial order) of *events* (*i.e.*, instances of operations) that modify the memory system state.[8] These events are organized according to a partial order relation. The following information is recorded for each event: its type (read, write, acquire, release), the agent that generated it (its *issuer*), and the location on which it was performed.

Events are temporally ordered by a relation $\prec$. Each processor must act as if it observed the events in an order compatible with $\prec$. When a processor performs an operation, an event is added to the history, and the $\prec$ relation is updated accordingly. In practice, the memory system does not maintain such a history, but this view is useful for thinking of consistency models in an implementation-independent way. How the relation $\prec$ is updated depends on the consistency model adopted. For instance, SC requires a total order of events, common to all processors. Therefore, each event would be ordered with respect to all events issued previously. In more relaxed models like LC, a partial order is sufficient.

For any memory model, a key question is: what value should be returned when a processor performs a read? For a strong memory consistency model, there is a unique value to be returned, which is the value written by the most recent write operation to that location. However, when a weaker memory model is used, there may be more than one value associated with a single location at a given time. In such models, a read operation is associated with a *set* of values that are considered readable for that location.

A specification of a memory consistency model can thus be characterized by two main features:

- What is the precedence relation between a new event and other events already recorded in the history?

[8] In fact, Gao and Sarkar speak only of *operations* and not *events*. Since any operation may be performed more than once, they define a history as a partially ordered *multiset* of operations. We find it more natural to distinguish between operations and events and use a partially ordered *set* of events.

In the case of the LC model, this question is answered as follows. A new write, acquire, or release event e by a processor agent p on a location ℓ is ordered so that it succeeds any event e already issued by p on ℓ. In other words, the set of events by p on ℓ is linearly ordered. Furthermore, since LC assumes that the history is a partial order, $\prec$ is transitive, so the new event also succeeds any event $e' \prec e$, possibly including events issued by other processor agents.

In the case of a new acquire event a, the partial order is updated further. The latest release event issued on ℓ (by any processor agent)[9] precedes a, along with any events that precede that release. This release could have been issued by any processor agent, not necessarily the issuer of the new acquire. Hence, it is through acquires that events issued by different processor agents are ordered according to $\prec$.

- Which values are associated with a new read event?

  In the LC model, when a processor agent p issues a read on a location ℓ, any write event on ℓ that has not been "overwritten" by another write event has its value associated with the new read event. We formalize this notion as follows. Let e be the last event issued by p; then according to the LC model, write event w is readable by p if and only if there is no write event w' such that $w \prec w' \preceq e$. This can be true of a write event w in either of the following ways:

  - If w precedes e and w is "recent" in the sense that there is no intervening write event between w and e, w's value is readable.
  - Alternatively, if w is simply unordered with respect to e, w's value is also readable.[10]

Our specification differs from Gao and Sarkar's description in a couple of aspects. First, we model a read as a single-step operation. In Gao and Sarkar's conception, a read first generates a read event and places it in the history, and then determines the readable values for the read. Our conception is more abstract, and it eliminates the need to place read events in the history. Second, our rules ensure that $\prec$ remains a transitive relation throughout the course of the system's execution. While Gao and Sarkar clearly intend for their precedence relation to be transitive, the state transformations they propose would not maintain that property.

LC_{mm}: Universes, attributes, and relations

We define universes ReadEvent, WriteEvent, AcquireEvent and ReleaseEvent to represent the sets of events of various types. We use the term Event to refer to the union of these universes.[11] Each Event has an issuer attribute that indicates which agent issued the event. In addition, a WriteEvent has a val attribute indicating the value written.

[9] Note that there is at most one latest release for ℓ at any given time, since (as pointed out in §2 and as formalized by Run Condition 3) a processor agent may only release ℓ if it has (exclusive) ownership of ℓ.

[10] Note that if w is unordered with respect to e, then the associated write has been performed by another processor agent q, and p and q have not synchronized with proper acquire/release operations. Thus the value of w could have been written to memory at an arbitrary moment, which is why it must be considered readable by p.

[11] In object-oriented terms, Event could be seen as a *general* or *base type* and the others as *specializations* or *subtypes*.

Function	Profile/Description
e.issuer	Event $\rightarrow$ ProcAgent $\cup$ InitAgent Agent that issued e.
w.val	WriteEvent $\rightarrow$ Value Value written at w.
p.latestEvent	ProcAgent $\rightarrow$ Event The most recent event issued by p.
ℓ.latestRelease	Location $\rightarrow$ ReleaseEvent The most recent release event issued on location ℓ.
i.initWrite	InitAgent $\rightarrow$ WriteEvent The write event issued by i.
reads?(rd, v)	ReadEvent $\times$ Value $\rightarrow$ Boolean Is v in the set of values read at event rd?
$e \prec e'$	Event $\times$ Event $\rightarrow$ Boolean Does e precede e' in the current system history?

Figure 7: Additional attributes and relations for LC_{mm}.

We introduce attributes to maintain the most recent events issued. Each processor agent has a latestEvent attribute that gives the most recent event issued by the agent. Similarly, each initializer agent has an initWrite attribute that gives the write event issued by the agent. Finally, each location has a latestRelease attribute that gives the most recent release event issued on the location.

Finally, we define two key relations:

- reads?(rd, v) indicates whether value v can be read at ReadEvent rd. The set of possible values that can be read by ReadEvent rd is thus the set $\{v|\text{reads?}(rd, v)\}$.
- $e \prec e'$ represents the partial order among memory events.

Initially, the relations $\prec$ and reads? are empty because there are no events in the system. Attributes and relations associated with events and with locations are presented in Figure 7.

Terminology

The following terms refer to the issuing of events in a run of LC_{mm}.

Definition *An event e with e.*issuer $= p$ *(for some* ProcAgent *or* InitAgent *p) is a p-*event.

Definition *If a* ProcAgent *or* InitAgent *p makes a move Rd that creates a* ReadEvent *rd, we say that p* issues a read event *rd at Rd. (Similarly for write, acquire, and release.)*

Definition *If a* ProcAgent *p reads at a move Rd and* readOK?(w, p) *for a* WriteEvent *w, we say that p* reads *w at Rd. We also say that p* reads value *w.*val *at Rd.*

LC_{mm}: Conditions on runs

We restrict attention to runs in which the following conditions are true in the initial state of LC_{mm}:

Init condition 4 *For every* Location ℓ, ℓ.latestRelease.undef?.
(Initially, there is no latest release event defined for any location.)

Init condition 5 *For every* ProcAgent p, p.latestEvent.undef?.
(Initially, there is no latest event defined for any processor agent.)

LC_{mm}: Terms and transition rules

The rules for write, acquire, and release operations by processor agents in the LC model are given in Figure 8. In each of these rules, a new event of the appropriate type is created using the **extend** construct. The issuer attribute of this event is updated to Self, which is the agent that executes the rule and generates the event.

The rule for read operations is given in Figure 9. The term readOK?(w, p), also defined in Figure 9, determines whether the write value of WriteEvent w is readable for ProcAgent p. For the value of WriteEvent w to be readable by processor agent p, w must be a write to the appropriate location, and there must be no WriteEvent w' that intervenes between w and the last event issued by p. Note that in the case where $w \prec p$.latestEvent does not hold (*i.e.*, there is no ordering relation between the write w and the latest event of p), there is no such w', so the value written by w will indeed be readable according to the LC model.

In the case of a read, the set of possible values that can be returned is associated with the new ReadEvent by updating the reads? relation. Any write event whose value is considered readable (according to readOK?) is in the set. For all non-read events,[12] the partial order relation $\prec$ is updated to account for the newly created event:

- The new event succeeds its issuer's latest event (as well as all predecessors of that event).
- Synchronization between processors imposes additional ordering constraints. In the LC model, these synchronizations occur exclusively through acquire and release operations. Thus a new AcquireEvent succeeds the latest release event on the location being acquired — which, by Run Conditions 2 and 3, is sure to exist and is sure to have been performed by the appropriate ProcAgent — as well as all predecessors of the latest release.

The rules for write and release operations by initializer agents (*InitWrite* and *InitRelease*, respectively) appear in Figure 10. In LC_{mm}, they are similar to their processor-agent counterparts (*Write* and *Release*, respectively). As we shall see in the next section, this is not the case for LC_{cp}: in that model, *InitWrite* differs significantly from *Write*, as does *InitRelease* from *Release*.

The rules presented in Figures 8–10 refine the processor agent and initializer agent modules of LC_0. Along with the ownership agent module in Figure 6, they complete LC_{mm}, the ASM representation of the LC model. In the next section, we describe LC_{cp}, the ASM representation of the LC cache protocol.

[12] As noted earlier, only non-read events are inserted into the partial order, contrary to [10].

rule *Write*:
extend WriteEvent **with** w
    w.issuer := Self
    w.val := Self.writeVal
    Order w *after* Self.latestEvent *and its predecessors*
    Self.latestEvent := w
Get next operation

rule *Acquire*:
if Self.loc.owner $\neq$ Self **then** Self.waiting? := true
else
    extend AcquireEvent **with** a
        a.issuer := Self
        Order a *after* Self.latestEvent *and its predecessors*
        Order a *after* Self.loc.latestRelease *and its predecessors*
        Self.latestEvent := a
    Get next operation

rule *Release*:
extend ReleaseEvent **with** r
    r.issuer := Self
    Order r *after* Self.latestEvent *and its predecessors*
    Self.latestEvent := r
    Self.loc.latestRelease := r
Self.loc.owner := undef
Get next operation

rule *Order* e *after* d *and its predecessors*:
if d.def? **then**
    $d \prec e$:= true
    do-forall c: Event: $c \prec d$
        $c \prec e$:= true

Figure 8: LC_{mm} rules for write, acquire and release operations.

rule *Read*:
extend ReadEvent **with** rd
    rd.issuer := Self
    do-forall w: WriteEvent: readOK?(w, Self)
        reads?(rd, w.val) := true
Get next operation

term readOK?(w, p):
w.issuer.loc = p.loc and not ($\exists w'$: WriteEvent) $w \prec w' \prec p$.latestEvent

Figure 9: Rule and auxiliary term for read operation in LC_{mm}.

rule *InitWrite*:
extend WriteEvent **with** w
    w.issuer := Self
    w.val := Self.writeVal
    Self.initWrite := w
Self.opType := release

rule *InitRelease*:
extend ReleaseEvent **with** r
    r.issuer := Self
    Self.initWrite $\prec r$:= true
    Self.loc.latestRelease := r
Self.opType := undef
Self.loc.initialized? := true

Figure 10: LC_{mm} rules for the initial write and release operations.

4 The LC cache protocol

In LC_{mm}, the state of the memory system is represented as a partially ordered set of events, and values are simply stored as attributes of events. We now present LC_{cp}, a formal model of the LC cache protocol, in which we make some stronger assumptions about how values are stored. In particular, we assume that each processor is equipped with its own cache, and that there is a set of memory areas collectively called *main memory*, distinct from any processor's cache. Each location has a unique value stored in main memory. Processors store values for reading and writing in their caches. When a processor writes to a location the new value is only written to the processor's cache. Eventually this value is *written back* to the main memory. Thus in this more concrete model, agents update cache entries and main memory locations instead of a history of events.

At any time, each cache entry is either *valid* or *invalid*, and a valid entry is either *clean* or *dirty*. A valid entry has a readable value, while an invalid one does not. A clean entry has a value from main memory that has not been overwritten; a dirty entry has a value written by the local processor that has not been written back to the main memory. When all the cache entries are occupied, a write or read of a location with no entry in the cache requires the removal (or *ejection*) of an existing location from the cache. A cache replacement policy is used to select which location should be removed from the cache; since we do not consider the details of any particular cache replacement policy, we use a monitored function to model the choice of which cache entry to eject.

In a multiprocessor machine, it is unrealistic to conceive of a writeback to main memory as a single-step action. There is an intrinsic delay between the *initiation* of a writeback (when the value stored in a cache entry is sent toward main memory) and the *completion* of the writeback (when the value is recorded in main memory). Writebacks may be completed concurrently with actions by processor agents, and a writeback to a remote main memory location is certainly not performed by the processor agent itself. To represent the process of writing back values to main memory, we introduce a universe of *writeback agents*. A writeback is initiated by generating a *writeback agent* and copying the dirty cached value to the writeback agent. The writeback is completed when the writeback agent copies this value to main memory.

An ejection removes a value from a processor agent's cache. Thus when a dirty entry is written back and ejected, it may be necessary for a processor agent to fetch a value from main memory. In many multiprocessor machines, however, a value may still be available locally even if the value has been ejected from the cache earlier. For instance, there may be a local *write buffer* that temporarily stores values that are to be written to main memory. If a processor agent performing a read finds no valid cache entry, it may still find a value in its write buffer that was recently sent to main memory. In this way, the processor agent avoids having to consult main memory.

Indeed, any implementation of the cache protocol *must* have a way for a processor agent to access its latest value in the transition period between ejection from the cache and writeback to main memory. Otherwise, a subsequent read may violate the memory model. Consider a simple uniprocessor example in which a processor agent performs two writes in succession, with no acquire or release actions. At some later time, it may be that the first write has been written back to main memory, and a writeback of the second write has been initiated but not yet completed. Imagine that the processor agent performs a read at this point. If it were to consult main memory for the read, it would get the value of the first write — a violation of the memory model.

The order in which writebacks are completed must also be restricted to avoid violating the memory model. For instance, in the previous example, if the two writebacks on the processor agent's cache entry are initiated but then completed in reverse order, the value in main memory

will come from the first writeback. If the processor agent were to read this value, the memory model would be violated. Gao and Sarkar do not make any explicit claims about the order of writebacks, so we will assume that writebacks must be completed in the order they were initiated.[13]

Our view of writebacks as multi-step actions requires us to clarify the meaning of a release operation. As noted earlier, one effect of a release is to make the last write by the releaser available to other processor agents. This is why a release initiates a writeback in the case of a dirty cache entry: the write that made the cache entry dirty must be propagated to main memory. But since a writeback cannot be performed in a single step, the following question arises: is it sufficient to *initiate* the writeback before completion of the release (*i.e.*, giving up ownership and proceeding to the next operation), or must the writeback also be *completed*? Gao and Sarkar [10] indicate the latter. This implies that a releasing processor agent may have to wait for a writeback to complete before proceeding to the next operation.[14] In fact, Gao and Sarkar require that all writebacks must be completed before the release can proceed.[15]

The actions for each operation are as follows. A read puts a value in the processor agent's cache, if there is no valid entry present in the cache already. This value comes from the processor agent's most recent writeback agent, if it is storing a value that has not been written back; otherwise it comes from main memory. A write generates a value, caches it, and updates the status of the cache entry to dirty. An acquire of a location invalidates the cache entry for the location, unless it is dirty (in which case the last value written by the processor remains in the cache, a legal value for subsequent read operations). A release of a dirty location initiates a writeback of the value stored in the cache, then waits until the value is transferred to main memory. Only when the writeback is completed does the release terminate.

Note that the LC cache protocol only requires two inexpensive operations to enable synchronization between multiple processors: the self-invalidation of cache entries that are not dirty for the Acquire rule, and the writeback of a dirty cache entry for the Release rule. Therefore no expensive invalidation or update requests need to be sent across the network under the LC cache protocol.[16]

LC_{cp}: Attributes

Like in the general model, in LC_{cp} a processor agent p is associated with a particular processor p and location ℓ. For each processor p, the attribute cacheVal gives the value in p's cache for location ℓ (if any such value exists), and the attributes cacheValid? and cacheDirty? give the valid/invalid status and dirty/non-dirty status of the cache entry.

In order not to tie our model to any specific cache replacement policy, the cache entry to be ejected (if any) is determined by a monitored function: the attribute ejectee. For each processor agent p, ejectee selects another processor agent associated with the same processor and which has

[13] There are undoubtedly looser restrictions possible; we choose this one for its simplicity.

[14] Later, they claim that "this wait for write completion does not require exclusive ownership". This would seem to suggest the contrary: a releasing processor agent may simply relinquish ownership, initiate a writeback, and continue to the next operation. However, they have a different intended meaning here for "exclusive ownership" [8]. They are referring here to exclusive ownership in the SC sense (where only the owner can perform reads or writes). They are simply emphasizing that other processor agents can perform reads and writes on a location while the owner of the location prepares to release.

[15] One can imagine a less stringent requirement: only the *latest* writeback must complete, and any earlier writebacks need not complete at all.

[16] Note that, in order to satisfy the LC model when a release operation is performed, any previous write must be completed, which thus incurs some additional cost (acknowledgement of writes).

Function	Profile/Description
p.cacheVal	ProcAgent → Value Value (if any) that p has cached.
p.cacheValid?	ProcAgent → Boolean Does p have a readable cached value?
p.cacheDirty?	ProcAgent → Boolean Has p performed a write that has not been written back to main memory yet?
p.ejectee	ProcAgent → ProcAgent Processor agent whose cache entry is to be ejected to make room for p's entry (monitored).
ℓ.MMVal	Location → Value Value of ℓ stored in main memory.
p.latestWB	ProcAgent → WritebackAgent Writeback agent most recently generated by p.
wb.issuer	WritebackAgent → ProcAgent Processor agent that generated wb.
wb.val	WritebackAgent → Value Value that wb is to write to main memory.
wb.active?	WritebackAgent → Boolean Does wb still have to write its value back to main memory?

Figure 11: Additional attributes for LC_{cp}.

a cache entry; the cache entry of p.ejectee is then to be ejected in order to make room for p's entry.

The attribute MMVal is associated with each location ℓ and represents the value currently stored in the main memory for ℓ. We introduce a universe WritebackAgent representing the agents charged with writing values to main memory. The function latestWB is an attribute associated with each ProcAgent, giving the writeback agent most recently generated by the processor agent. We associate three attributes with the WritebackAgent universe: issuer, which gives the processor agent that generated the writeback agent; val, which gives the value to write to main memory; and active?, which determines whether a given writeback agent has yet to write its value to main memory. Figure 11 summarizes the attributes that we use to model caches, writeback agents, and the main memory.

Terminology

In LC_{cp}, releases generally are multi-step actions. Therefore, we must reformulate what it means for a processor agent to perform a release. In our terms, a processor agent first *prepares* to perform a release by initiating a writeback of its dirty cache entry and waiting for the writeback to complete. It only *performs* the release (relinquishing ownership) after these actions have completed. We formalize this as follows.

Definition *If a* ProcAgent p *makes a move* PR_p *at which* p.opType = release *and* (p.cacheDirty? or p.latestWB.active?)*, we say that* p prepares to release *at* PR_p.

Definition *If a* ProcAgent p *makes a move* R_p *at which* p.opType = release *and* not (p.cacheDirty? or p.latestWB.active?)*, we say that* p releases *at* R_p.

We use the following terms to characterize read actions and cache maintenance actions in ρ_{cp}.

Definition *If a* ProcAgent p *reads at a move* Rd_p*, we say that* p reads value v *at* Rd_p*, where*

- $v = p$.cacheVal *if* p.cacheValid?*;*
- *otherwise,* $v = p$.latestWB.val *if* p.latestWB.active?*;*
- *otherwise,* $v = p$.loc.MMVal.

Definition *Let* p *be a* ProcAgent *that reads at a move* Rd_p.

- *If* not p.cacheValid? *and* not p.latestWB.active?*, we say that* p *performs a* miss read *at* Rd_p*;*
- *Otherwise, if* p.cacheDirty? *or* p.latestWB.active?*, we say that* p *performs a* dirty read *at* Rd_p*;*
- *Otherwise, we say that* p *performs a* clean read *at* Rd_p.

Definition *Let* p *be a* ProcAgent*, and let* wb_p *be a* WritebackAgent *for which* wb_p.issuer = p.

- *If at a move* I_p*,* p.cacheDirty? *is updated from* true *to* false*, we say that a* writeback *of* p*'s cache entry is* initiated *at* I_p.[17]

[17] Note that a writeback may be initiated by p itself (through a release) or by another ProcAgent (through a read or write that triggers an ejection of p's cache entry).

- *If at a move C_p, wb_p.*active? *is updated from* true *to* false*, we say that a* writeback *of p's cache entry is* completed *at C_p.*
- *Let I_p be a move at which a writeback of p's cache entry is initiated and wb_p is generated. Let C_p be a move of wb_p at which a writeback of p's cache is completed. Then we say that* the writeback initiated at I_p is completed at C_p.

LC_{cp}: Conditions on runs

We put the following restrictions on initial states of LC_{cp}.

Init condition 6 *For every* ProcAgent *p,* not (*p*.cacheValid? or *p*.cacheDirty?).
(Initially, each cache entry must be invalid and (therefore) non-dirty.)

Init condition 7 *The* WritebackAgent *universe is empty.*
(Initially, there are no values to write back to main memory.)

The attribute ejectee must take on reasonable values during a run. We restrict attention to runs that obey the following conditions:

Run condition 4 *For every* ProcAgent *p, if p*.ejectee.def?*, then p*.ejectee.proc = *p*.proc *and* *p*.ejectee.cacheValid?.
(The ejectee for a processor agent p must be a ProcAgent associated with the same processor as p and must have a valid cache entry.)

Run condition 5 *For every* ProcAgent *p, if p*.ejectee.def?*, then p*.ejectee.opType ≠ read *and* *p*.ejectee.opType ≠ write.
(The ejectee for a given processor agent must not currently be reading or writing, since otherwise it would need its cache entry.)

Finally, we ensure that writebacks are completed in the order in which they are initiated.

Run condition 6 *Let p be a* ProcAgent*. Let I_p be a writeback initiation on p's cache entry that is completed at C_p. Let I'_p be a writeback initiation on p's cache entry that is completed at C'_p. If I_p precedes I'_p, then C_p precedes C'_p.*

LC_{cp}: Transition rules

The rules and terms associated with cache ejection and writeback are presented in Figure 12. The ejection of a cache entry requires an invalidation of the cache entry, and a writeback if the entry is dirty. The writeback initiation updates the cache entry's status to non-dirty, generates a writeback agent, and passes the cached value to the writeback agent. The writeback agent module is simple: a writeback agent makes a single move in which it copies its value to main memory.

The rules for read, write, acquire, and release operations by processor agents are presented in Figure 13. If there is no valid cache entry, reading involves fetching a value from the last writeback agent, as represented by the update rule Self.cacheVal := Self.latestWB.val, or from main memory, as represented by the update rule Self.cacheVal := Self.loc.MMVal. Writing involves storing a new value in the cache, as represented by Self.cacheVal := Self.writeVal.

rule *Eject cache entry of* p:
p.cacheValid? := false
if p.cacheDirty? **then**
    Initiate writeback on cache entry of p

rule *Initiate writeback on cache entry of* p:
p.cacheDirty? := false
extend WritebackAgent **with** wb_p
    wb_p.val := p.cacheVal
    wb_p.active? := true
    p.latestWB := wb_p

module WritebackAgent:
if Self.active? **then**
    Self.loc.MMVal := Self.val
    Self.active? := false

term p.allWritebacksCompleted?:
($\forall wb_p$: WritebackAgent: wb_p.proc = p) not wb_p.active?

Figure 12: LC_{cp} rules for cache maintenance.

In the case of a read or write, a new cache entry may be needed; therefore the attribute ejectee is checked to determine whether a cache entry is to be ejected to make room for the new one. The rules for acquire and release operations are simple. An acquire invalidates a clean cache entry. A release initiates a writeback of the cache entry, if it is dirty. Only when all writebacks on the cache entry are completed does the release terminate.

The rules for write and release operations by initializer agents are given in Figure 14. The initial write records the initializer's value directly to main memory. As a result, the initial release is not really needed to force the value to main memory.

5 LC_{cp} obeys LC_{mm}

In this section, we show that the cache protocol described by LC_{cp} implements the abstract model described by LC_{mm}. In particular, we show that any value read in an execution of LC_{cp} is also a legal value in an equivalent execution of LC_{mm}. In a run of LC_{mm}, for each read operation a *set* of legal readable values is computed, while in the run of LC_{cp} a *single* value is read at each read operation. We consider runs of LC_{mm} and LC_{cp} in which the memory operations (read, write, acquire, release) that are performed and the order in which they are performed are identical. We then show that for each read operation of LC_{cp}, the single value read is in the set of readable values computed at the corresponding move of LC_{mm}'s run.

rule *Read*:
if not Self.cacheValid? **then**
    if Self.allWritebacksCompleted? **then** Self.cacheVal := Self.loc.MMVal
    else Self.cacheVal := Self.latestWB.val
    Self.cacheValid? := true
    if Self.ejectee.def? **then** *Eject cache entry of* Self.ejectee
Get next operation

rule *Write*:
Self.cacheVal := Self.writeVal
Self.cacheValid? := true
Self.cacheDirty? := true
if Self.ejectee.def? **then** *Eject cache entry of* Self.ejectee
Get next operation

rule *Acquire*:
if Self.loc.owner $\neq$ Self **then** Self.waiting? := true
else
    if Self.cacheValid? and not Self.cacheDirty? **then** Self.cacheValid? := false
    Get next operation

rule *Release*:
if Self.cacheDirty? **then** *Initiate writeback on cache entry of* Self
elseif Self.allWritebacksCompleted? **then**
    Self.loc.owner := undef
    Get next operation

Figure 13: LC_{cp} rules for read, write, acquire and release operations by processor agents.

rule *InitWrite*:
Self.loc.MMVal := Self.writeVal
Self.opType := release

rule *InitRelease*:
Self.opType := undef
Self.loc.initialized? := true

Figure 14: LC_{cp} rules for write and release operations by initializer agents.

Equivalent runs of LC_{mm} and LC_{cp}

We start by considering what it means for runs of LC_{mm} and LC_{cp} to be "equivalent". An ASM *run* consists of a partial order of *moves* performed by agents, with some agent executing its associated module at each move. Informally, for runs of the two models to be equivalent, the system components (locations and processors) must be the same, and the same agents must make the same moves in the same order. More precisely, the following conditions must be met:

- All the static information (*e.g.*, number of processors, locations and agents) must be the same in the two runs.
- The runs must have the same partial order of moves.
- For each move, the environment in the two runs must produce the same results for the functions nextOpType, writeVal, and nextOwner.

We formalize the above as follows:

- Let σ be a state of LC_{mm} or LC_{cp}. Then σ^- is the reduct of σ to the static and monitored functions common to LC_{mm} and LC_{cp} (*i.e.*, the static and monitored functions of LC_0, introduced in §2).
- A state σ_{mm} of LC_{mm} is *equivalent* to a state σ_{cp} of LC_{cp} if σ^-_{mm} and σ^-_{cp} are isomorphic.

Let $\rho^*_{cp} = (\mu^*_{cp}, \alpha_{cp}, \sigma^*_{cp})$ be a run of LC_{cp}. μ^*_{cp} is a partially ordered set (poset) of moves, α_{cp} is a function mapping moves to agents, and σ^*_{cp} is a function mapping finite initial segments of moves to states of LC_{cp}. α_{cp} gives the agent performing each move, and σ^*_{cp} gives the state resulting from each finite initial segment of moves.

In our proofs, we find it simpler to consider a sequential "equivalent" of a distributed run than the run itself. As mentioned by Gurevich and Rosenzweig [14], a sequential run has two attractive properties: each move can be seen as executing in a unique, well-defined state, and changes to monitored functions can be seen as located in time (essentially, allowing monitored changes to be represented as actions by hidden agents). According to the ASM Lipari guide [12], we lose no generality by proving correctness of an arbitrary *linearization* of a run. Hence we consider an arbitrary linearization $\rho_{cp}= (\mu_{cp}, \alpha_{cp}, \sigma_{cp})$ of ρ^*_{cp}. μ_{cp} is a linearly ordered set (*i.e.*, sequence or interleaving) of moves that has exactly the same moves as μ^*_{cp} and preserves all the ordering of μ^*_{cp}. Since μ_{cp} is a sequence, every finite initial segment of μ_{cp} is a prefix of μ^*_{cp}, so σ_{cp} is a restriction of σ^*_{cp} to finite prefixes of μ^*_{cp}.

Let $\rho_{mm}= (\mu_{mm}, \alpha_{mm}, \Sigma_{mm})$ be a run of LC_{mm} that is *equivalent* to LC_{cp}, as defined below. Informally, in the runs ρ_{mm} and ρ_{cp} the same agents perform the same operations in the same order; only the implementation details differ. In LC_{mm} the partial order $\prec$ is updated, while in LC_{cp} it is the cache entries and main memory locations that are updated.

Fewer moves are made in ρ_{mm} than in ρ_{cp}. First, WritebackAgents do not exist in ρ_{mm} and so do not make moves. Second, a release in ρ_{mm} is always a single-move action; there is no need to prepare for a release. We restrict μ_{mm} to the moves of μ_{cp} that are neither writeback-agent moves nor release preparation moves. More formally, $\mu_{mm} = \mu_{cp} \setminus (\{M\text{: WritebackAgent}(\alpha_{cp}(M))\} \cup \{M{:}\alpha_{cp}(M) \text{ prepares to release at } M\})$. Likewise, we define α_{mm} as the restriction of α_{cp} to moves of μ_{mm}. Finally, for each prefix X of μ_{mm}, $\sigma_{mm}(X)$ is equivalent to $\sigma_{cp}(X)$. Since the only sources

of nondeterminism in LC_{mm} are the monitored functions nextOpType, writeVal and nextOwner, and these are identical in ρ_{cp} and ρ_{mm}, ρ_{mm} is unique up to isomorphism.

Lemmata: ordering of events in ρ_{mm}

An inspection of the rules in LC_{mm} shows that the predecessors of a given event e are computed at the time the event is issued, through firing of a rule of the form *Order e after ... and its predecessors*. Note that the firing of this rule does not add predecessors to any events other than the new event e. Furthermore, an inspection of the rules of LC_{mm} shows that predecessors are never removed: once $d \prec e$ is updated to true for any two events d and e, $d \prec e$ never becomes false later. Thus the set of predecessors of a given event is established at the time of its creation and never changes. We formalize this notion as follows.

Claim 1 *Let d and e be* Event*s, and let E be the move at which e is issued.*

- *If $\prec$ is updated at E so that $d \prec e$, then $d \prec e$ forever after E.*
- *If $\prec$ is not updated at E so that $d \prec e$, then* not *$d \prec e$ forever after E.*

Thus it makes no difference whether we speak of $d \prec e$ at some *particular* move after E or at *all* moves after E. We use the following abbreviatory device: we take "$d \prec e$" to mean "$d \prec e$ forever after E, the move at which e is issued".

Our first lemma asserts that $\prec$, in accordance with Gao and Sarkar's intentions, is a transitive relation.

Lemma 1 *Let e and f be* Event*s such that $e \prec f$. Then for any* Event *g such that $f \prec g$, it is also the case that $e \prec g$.*

Proof. Let p_f and p_g be the issuers of f and g, respectively. Let E_f and E_g be the moves at which f and g are issued, respectively. We induct on the number of successors of f issued before E_g.

- If no such successors are issued before E_g, then at E_g, f is a maximal Event (*i.e.*, an Event with no successors). By inspection of the rules of LC_{mm}, there are only two ways in which g can become a successor of f:
  - $p_f = p_g$ and $f = p_g$.latestEvent at E_g;
  - g is an AcquireEvent and $f = p_g$.loc.latestRelease at E_g.

  In either case, the rule *Order g after f and its predecessors* fires at E_g, updating $\prec$ so that not only $f \prec g$ but also $e \prec g$.
- Otherwise, assume that for every successor e_{succ} of f issued before E_g, $e \prec e_{succ}$. Then there are only two ways in which g can become a successor of f:
  - $f \prec p_g$.latestEvent at E_g. Then since p_g.latestEvent is a successor of f issued before E_g, by the inductive hypothesis, $e \prec p_g$.latestEvent. At E_g, the rule *Order g after p_g.*latestEvent *and its predecessors* fires, updating $\prec$ so that not only $f \prec g$ but also $e \prec g$.

– g is an AcquireEvent and $f \prec p_g.\mathsf{loc.latestRelease}$ at E_g. Then since $p_g.\mathsf{loc.latestRelease}$ is a successor of f issued before E_g, by the inductive hypothesis, $e \prec p_g.\mathsf{loc.latestRelease}$. At E_g, the rule *Order g after p_g.*loc.latestRelease *and its predecessors* fires, updating $\prec$ so that not only $f \prec g$ but also $e \prec g$.

Next, we prove some basic properties of the ordering of events in ρ_{mm}. Lemma 2 states that the events issued by a ProcAgent or InitAgent are linearly ordered by $\prec$; whenever a processor agent issues two events d and e in sequence, d becomes a predecessor of e.

Lemma 2 *In ρ_{mm}, let p be a* ProcAgent*, let d_p be a p-event issued at a move D_p, and let e_p be a p-event issued at a move E_p after D_p. Then $d_p \prec e_p$.*

Proof. We induct on the number of p-events issued between D_p and E_p.

- If there are no such events issued, then $p.\mathsf{latestEvent} = d_p$ at E_p, so at E_p the rule *Order e after d and its predecessors* fires, updating $\prec$ so that $d_p \prec e_p$.
- Otherwise, let $last_p$ be the last p-event issued before E_p, at a move $Last_p$ in the interval (D_p, E_p). By the inductive hypothesis, we assume that $d_p \prec last_p$. At $Last_p$, $p.\mathsf{latestEvent}$ is updated to $last_p$ and is not updated in the interval $(Last_p, E_p)$, so $p.\mathsf{latestEvent} = last_p$ at E_p. The rule *Order e_p after $last_p$ and its predecessors* fires at E_p, updating $\prec$ so that $last_p \prec e_p$. By Lemma 1, $d_p \prec e_p$.

□

Lemma 3 states that release events issued on a common location are linearly ordered by $\prec$.

Lemma 3 *In ρ_{mm}, let p and q be* ProcAgent*s or* InitAgent*s for which $p.\mathsf{loc} = q.\mathsf{loc}$. Let R_p be a move at which p issues a release event r_p, and let R_q be a move after R_p at which q issues a release event r_q. Then $r_p \prec r_q$.*

Proof. Let $\ell = p.\mathsf{loc} = q.\mathsf{loc}$. By Run Condition 3, q can only release at R_q if q acquires in the interval (R_p, R_q). Let A_q be the move at which the last acquire by q occurred, and let a_q be the acquire event issued at A_q. We induct on the number of release actions on ℓ between R_p and A_q.

- If no release actions on ℓ are performed in this interval, then at R_p, $\ell.\mathsf{latestRelease}$ is updated to r_p and is not updated in the interval (R_p, A_q). Thus at A_q, $\ell.\mathsf{latestRelease} = r_p$, so q updates $\prec$ at A_q so that $r_p \prec a_q$.
- Otherwise, let r_{last} be the last release event on ℓ issued before R_q, at a move R_{last} in the interval (R_p, R_q). By the inductive hypothesis, we assume that $r_p \prec r_{last}$. By Run Condition 3, R_{last} precedes A_q. At R_{last}, $\ell.\mathsf{latestRelease}$ is updated to r_{last} and is not updated in the interval (R_{last}, A_q), so $\ell.\mathsf{latestRelease} = r_{last}$ at A_q. The rule *Order a_q after r_{last} and its predecessors* fires at A_q, updating $\prec$ so that $r_{last} \prec a_q$. By Lemma 1, $r_p \prec a_q$.

In either case, we have that $r_p \prec a_q$. By Lemma 2, $a_q \prec r_q$, so by Lemma 1, $r_p \prec r_q$. □

The following lemma concerns how events issued by different agents in ρ_{mm} can become ordered with respect to each other. This is important in determining whether a write event w_p by one agent

p is readable by another agent q (where p and q operate on a common location). In particular, it is necessary to determine whether w_p precedes q's latest event (according to $\prec$). If not, w_p is readable; if so, w_p is only readable if there is no write event intervening between w_p and q's latest event. Lemma 4 asserts that

- a p-write becomes a predecessor of a q-event if p releases after the write and q then acquires;
- this is the *only* way that a p-write can come to be ordered with respect to a q-event.

Lemma 4 *In ρ_{mm}, let p be a* ProcAgent *or* InitAgent*, and let q be a* ProcAgent *for which $p \neq q$ but $p.\mathsf{loc} = q.\mathsf{loc}$. Let W_p be a move at which p issues a* WriteEvent *w_p, and let Rd_q be a move after W_p at which q reads. Then $w_p \prec q.\mathsf{latestEvent}$ at Rd_q if and only if*

- *p issues a* ReleaseEvent *r_p at a move R_p in the interval (W_p, Rd_q) and*
- *q issues an* AcquireEvent *a_q at a move A_q in the interval (R_p, Rd_q).*

Proof. Let $\ell = p.\mathsf{loc} = q.\mathsf{loc}$.

(if) By Lemma 2, $w_p \prec r_p$. Let R_{last} be the move of the last release on ℓ in the interval $[R_p, A_q)$. At R_{last}, a ReleaseEvent r_{last} is created, and $\ell.\mathsf{latestRelease}$ is updated to r_{last}. By Lemma 3, $r_p \prec r_{last}$, so by Lemma 1, $w_p \prec r_{last}$. In the interval (R_{last}, A_q), $\ell.\mathsf{latestRelease}$ is not updated, so $\ell.\mathsf{latestRelease} = r_{last}$ at A_q. Thus $r_{last} \prec a_q$, so by Lemma 1, $w_p \prec a_q$. By Lemma 2, $a_q \prec q.\mathsf{latestEvent}$ at Rd_q, so by Lemma 1, $w_p \prec q.\mathsf{latestEvent}$ at Rd_q.

(only if) Let A_q be the first move in the interval (W_p, Rd_q) at which a q-event is issued and made a successor of w_p; call the new q-event a_q. If a_q is a WriteEvent or ReleaseEvent, q fires the rule *Order a_q after $q.\mathsf{latestEvent}$ and its predecessors*, updating $\prec$ so that a_q succeeds only the q-event $q.\mathsf{latestEvent}$ and its predecessors. But since w_p precedes no q-event at A_q, a_q cannot be a WriteEvent or ReleaseEvent; it must be an AcquireEvent. So q must acquire at A_q.

Let A_s be the first move in the interval $(W_p, A_q]$ at which w_p is made a predecessor of an s-event for any ProcAgent $s \neq p$. By Claim 1, s must issue an event a_s at A_s. If a_s is a WriteEvent or ReleaseEvent, s fires the rule *Order a_s after $s.\mathsf{latestEvent}$ and its predecessors*, updating $\prec$ so that a_s succeeds only the s-event $s.\mathsf{latestEvent}$ and its predecessors. Since w_p precedes no s-event at A_s, a_s cannot be a WriteEvent or ReleaseEvent; it must be an AcquireEvent. Therefore at A_s, it must be the case that s acquires and $w_p \prec \ell.\mathsf{latestRelease}$. A_s is the first move at which w_p becomes a predecessor of a non-p-event, so at A_s, any successor of w_p (including $\ell.\mathsf{latestRelease}$) must have been issued by p. Therefore $\ell.\mathsf{latestRelease.issuer} = p$. This is only possible if p releases at a move R_p in the interval (W_p, A_s).

□

Lemmata: Properties of ρ_{cp}

In Gao and Sarkar's description of the cache protocol, a cache entry can be in one of three states: invalid (*i.e.*, not valid, not dirty), clean (*i.e.*, valid, not dirty), or dirty (*i.e.*, valid, dirty). We must show that the (not valid, dirty) state never arises in our model. Initially, the cacheValid? and cacheDirty? attributes evaluate to false for all processor agents. An inspection of the rules of LC_{cp}

shows that cacheDirty? is updated to true only in a write, when cacheValid? is also updated to true. Furthermore, cacheValid? is updated to false only in an ejection, in which case cacheDirty? is either already false or updated to false. Therefore we can make the following claim.

Claim 2 *At every move of ρ_{cp} in which p.*cacheDirty? *for a* ProcAgent *p, it is also the case that p.*cacheValid?*.*

Lemma 5 states that when a processor agent performs a release, any write it has performed previously has been written back.

Lemma 5 *In ρ_{cp}, let W_p be a write by a* ProcAgent *p, followed by a release R_p by p. Then a writeback of p's cache entry is initiated and then completed in the interval (W_p, R_p).*

Proof. At W_p, p.cacheDirty? is updated to true, but at R_p, not p.cacheDirty?, so there must be a writeback of p's cache entry initiated in (W_p, R_p) to update p.cacheDirty? to false. Let I_p be the first such writeback initiation, and let wb_p be the WritebackAgent generated. At I_p, wb_p.active? is updated to true, but at R_p, not wb_p.active?, so in (I_p, R_p) there must be a writeback by wb_p to update wb_p.active? to false. □

Lemmata: properties of read operations

Lemmata 6–8 concern the three types of read operation in ρ_{cp}: dirty, miss, and clean. For a read operation of any type in ρ_{cp}, it is established that the value read is one of the (possibly many) values read at the corresponding move of ρ_{mm}.

Lemma 6 *In ρ_{cp}, let Rd_p^D be a move at which a* ProcAgent *p performs a dirty read of a* Value *v. Then in ρ_{mm}, p also reads v at Rd_p^D.*

Proof. First, consider ρ_{cp}. We start by tracing the value v back to a particular write W_q. There are two circumstances under which a read can be dirty:

- At Rd_p^D, p.cacheDirty?. In this case, p reads from its cache, so p.cacheVal $= v$ at Rd_p^D. Initially, not p.cacheDirty?, so p must write before Rd_p^D, updating p.cacheDirty? to true and updating p.cacheVal. Let W_p be the last such write.

  We show that the value written at W_p is v. There is no writeback of p's cache entry initiated in the interval (W_p, Rd_p^D), since it would update p.cacheDirty? to false and there is no write by p in (W_p, Rd_p^D) to update p.cacheDirty? back to true. Since p.cacheDirty? is (W_p, Rd_p^D), there is no ejection of p's cache entry in (W_p, Rd_p^D), so p.cacheValid? in this interval. Therefore any read by p in the interval (W_p, Rd_p^D) leaves p.cacheVal unchanged. Since p does not write in the interval (W_p, Rd_p^D), p.cacheVal is unchanged in the interval, so the value written at W_p is v.

- At Rd_p^D, not p.cacheValid?, but p.latestWB.active?. In this case, p reads a value from its last writeback agent, so at Rd_p^D, p.latestWB.val $= v$. Let wb_p be p.latestWB at Rd_p^D. Before Rd_p^D there is a writeback initiation I_p of p's cache entry, generating wb_p and updating wb_p.active? to true, and an ejection of the entry (updating p.cacheValid? to false), but the writeback is

not completed before Rd_p^D (since its completion would update wb_p.active? to false). Let W_p be the last write by p before I_p. (Since p.cacheDirty? at I_p, there must be such a write.)

We show that the value written at W_p is v. At I_p, wb_p.val is updated to p.cacheVal and is unchanged in (I_p, Rd_p^D), so p.cacheVal $= v$ at I_p. In (W_p, I_p), p.cacheDirty?, so by Claim 2, p.cacheValid?; hence any read by p in (W_p, I_p) leaves p.cacheVal unchanged. Thus the value written at W_p is v.

We now show that p does not write in (W_p, Rd_p^D). Assume to the contrary that p does write at W_p^* in (W_p, Rd_p^D). Clearly, W_p^* is not in (W_p, I_p), since W_p is p's last write before F_p. Then W_p^* is in (I_p, Rd_p^D). At W_p^*, p.cacheDirty? is updated to true, but at Rd_p^D, not p.cacheDirty?, so there must be a writeback of p's cache entry at I_p^* in (W_p^*, Rd_p^D) to update p.cacheDirty? to false. But at Rd_p^D, p.latestWB $= wb_p$, and a writeback initiation at I_p^* would update p.latestWB to a value other than wb_p. So there can be no such I_p^*.

Since p does not write in (W_p, Rd_p^D), the value written at W_p is v.

Now consider ρ_{mm}. We show that in this run, p also reads v at Rd_p^D. Let w_p be the WriteEvent issued at W_p; then w_p.val $= v$. Since p does not write in the interval (W_p, Rd_p^D), at Rd_p^D there is no p-WriteEvent $w_p^* \succ w_p$. Then not readOK?(w_p, p) at Rd_p^D only if, for some $q \neq p$, there is a q-WriteEvent w_q such that $w_p \prec w_q \prec p$.latestEvent. By Lemma 4, this is the case only if p releases and then q acquires in the interval (W_p, Rd_p^D).

Returning to ρ_{cp} we show that p does not release in (W_p, Rd_p^D). Assume to the contrary that p does release at R_p^* in (W_p, Rd_p^D).

- If p reads from its cache, then at Rd_p^D, p.cacheDirty?, but at R_p^*, not p.cacheDirty?, so there must be a write in (R_p^*, Rd_p^D) to update p.cacheDirty? back to true. But since there is no such write, there can be no such R_p^* in (W_p, Rd_p^D).
- If p reads from its last writeback agent, then at W_p, p.cacheDirty? is updated to true, and there is no writeback of p's cache entry initiated in (W_p, I_p) to update p.cacheDirty? to false, so p.cacheDirty? in $(W_p, I_p]$. Since not p.cacheDirty? at R_p^*, R_p^* is not in $(W_p, I_p]$. At I_p, wb_p.active? is updated to true, and there is no writeback completion by wb_p in (I_p, Rd_p^D), so wb_p.active? in $(I_p, Rd_p^D]$. Therefore, R_p^* is not in (I_p, Rd_p^D). Hence R_p^* can occur nowhere in (W_p, Rd_p^D).

Finally, we return to ρ_{mm}Ṡince p does not release in (W_p, Rd_p^D), readOK?(w_p, p) at Rd_p^D, so p reads v. □

Lemma 7 *In ρ_{cp}, let Rd_p^M be a move at which a* ProcAgent *p performs a miss read of a* Value *v. Then in ρ_{mm}, p also reads v at Rd_p^M.*

Proof. Let ℓ be p.loc. First, consider ρ_{cp}. At Rd_p^M, ℓ.MMVal $= v$ and not p.cacheValid?. We start by tracing the value v back to a particular write W_q. There are two cases:

- If there is a writeback to ℓ completed before Rd_p^M, let C_q be the move at which the last such writeback was completed, let wb_q be the WritebackAgent that performs the writeback, and

let q be the ProcAgent for which $wb_q.\mathsf{proc} = q$. At C_q, $\ell.\mathsf{MMVal}$ is updated to $wb_q.\mathsf{val}$, and $\ell.\mathsf{MMVal}$ is unchanged in the interval (C_q, Rd_p^M) (since C_q is the last writeback to ℓ before Rd_p^M), so $wb_q.\mathsf{val} = v$ at C_q.

Let I_q be the writeback initiation that generates wb_q. At I_q, $wb_q.\mathsf{val}$ is updated to $q.\mathsf{cacheVal}$. In (I_q, C_q), $wb_q.\mathsf{val}$ is not updated, so $q.\mathsf{cacheVal} = v$ at I_q. Let W_q be the last write by q before I_q. (Since $q.\mathsf{cacheDirty?}$ at I_q, there must be such a write.) In the interval (W_q, I_q), $q.\mathsf{cacheDirty?}$ and so by Claim 2, $q.\mathsf{cacheValid?}$; hence any read by q in (W_q, I_q) leaves $q.\mathsf{cacheVal}$ unchanged. Since there is no write by q in (W_q, I_q), the value written at W_q is v.

- If there is no writeback to ℓ before Rd_p^M, let q be the InitAgent of ℓ, let W_q be the write by q, and let C_q be the release by q. Since there is no writeback to ℓ in the interval (W_q, Rd_p^M), $\ell.\mathsf{MMVal}$ is unchanged in this interval, so the value written at W_q is v.

Now consider ρ_{mm}. We show that in this run, p also reads v at Rd_p^M. Let w_q be the WriteEvent issued at W_q; then $w_q.\mathsf{val} = v$. We show that $\mathsf{readOK?}(w_q, p)$ at Rd_p^M. For $\mathsf{readOK?}(w_q, p)$ to be false at Rd_p^M, there must be a WriteEvent w_s^* such that $w_q \prec w_s^* \preceq p.\mathsf{latestEvent}$. Assume that such a w_s^* exists; let s be the agent issuing w_s^*, and let W_s^* be the move at which s issues w_s^*. We have the following cases:

- $q = s = p$ (*i.e.*, p issues w_q and then issues another WriteEvent w_s^* before Rd_p^M). For readability, let $W_p = W_q$, $W_p^* = W_s^*$, $I_p = I_q$, $C_p = C_q$, and $wb_p = wb_q$. Then W_p^* must be in the interval (W_p, Rd_p^M).

  Returning to ρ_{cp} we show that W_p^* cannot exist. Assume to the contrary that p does write at W_p^* in (W_p, Rd_p^M). Since there is no write by p in (W_p, I_p), W_p^* must be in (I_p, Rd_p^M). Such a write would update $p.\mathsf{cacheDirty?}$ to true, but not $p.\mathsf{cacheDirty?}$ at Rd_p^M. So there must be a writeback of p's cache entry initiated in the interval (W_p^*, Rd_p^M), updating $p.\mathsf{cacheDirty?}$ to false. Let I_p^* be the last such writeback initiation, generating a WritebackAgent wb_p^*. Then $wb_p^* = p.\mathsf{latestWB}$ at Rd_p^M. At I_p^*, $wb_p.\mathsf{active?}$ is updated to true, but at Rd_p^M, not $wb_p.\mathsf{active?}$, so wb_p must write back at C_p^* in the interval (I_p^*, Rd_p^M), to update $wb_p.\mathsf{active?}$ back to false. Since I_p precedes I_p^*, C_p^* is in the interval (C_p, Rd_p^M) (by Run Condition 6). But since C_p is the last writeback to ℓ before Rd_p^M, this is impossible.

- $q = s \neq p$ (*i.e.*, q issues another WriteEvent w_s^* before Rd_p^M). Let $W_q^* = W_s^*$, and let $w_q^* = w_s^*$. Since $w_q^* \prec p.\mathsf{latestEvent}$ at Rd_p^M, by Lemma 4 q releases and then p acquires in the interval (W_q^*, Rd_p^M).

  Returning to ρ_{cp} we show that W_q^* cannot exist. Assume to the contrary that q does write at W_q^* in (W_q, R_q). By Lemma 5, there must be a writeback of q's cache entry initiated in (W_q, R_q). Since I_q is the first such writeback initiation after W_q, I_q must be in (W_q, R_q). Since W_q is q's last write before I_q, W_q^* must be in (I_q, R_q). By Lemma 5, there must be a writeback initiation of q's cache entry at I_q^* in (W_q^*, R_q), followed by a completion at C_q^* in (I_q^*, R_q). Since I_q precedes I_q^*, C_q^* must be in (C_q, R_q) (by Run Condition 6). But this is impossible, since C_q is the completion of the last writeback to ℓ before Rd_p^M.

- $q \neq s = p$ (*i.e.*, p issues a WriteEvent w_s^* before Rd_p^M). Let $W_p^* = W_s$. Since $w_q \prec w_p^*$ at Rd_p^M, by Lemma 4 q releases and then p acquires in the interval (W_q, W_p^*).

  Returning to ρ_{cp} we show that W_p^* cannot exist. Assume to the contrary that p does write at W_p^* in (A_p, Rd_p^M).

  First, we show that C_q must be in the interval (W_q, R_q). By Lemma 5, there must be a writeback of p's cache entry initiated and completed in the interval (W_q, R_q). Since I_q is the first writeback initiation of q's cache entry after W_q and C_q is the corresponding writeback completion, it must be that C_q is in (W_q, R_q).

  Next, we show that C_q must be in the interval (W_p^*, Rd_p^M). At W_p^*, p.cacheDirty? is updated to true, but at Rd_p^M, not p.cacheDirty?, so there must be a writeback initiation of p's cache entry in (W_p, Rd_p^M) to update p.cacheDirty? to false. Let I_p^* be the last such writeback initiation, and let wb_p^* be the WritebackAgent generated. Then at Rd_p^M, p.latestWB $= wb_p^*$. At I_p^*, wb_p^*.active? is updated to true, but at Rd_p^M, not wb_p^*.active?, so wb_p^* must write back in (I_p^*, Rd_p^M). Since C_q is the completion of the writeback to ℓ before Rd_p^M, C_q must be in (I_p^*, Rd_p^M).

  But since R_q precedes W_p^*, C_q cannot be in both (W_q, R_q) and (W_p^*, Rd_p^M).

- $q \neq s \neq p$ (*i.e.*, some ProcAgent other than p or q issues a WriteEvent w_s^* before Rd_p^M). Since $w_q \prec w_s^*$, by Lemma 4, q releases and then s acquires in the interval (W_q, W_s^*). Since $w_s^* \prec p$.latestEvent at Rd_p^M, by Lemma 4, s releases and then p acquires in the interval (W_s^*, Rd_p^M).

  Returning to ρ_{cp} we show that W_s^* cannot exist. Assume to the contrary that s does write at W_s^* in (A_s, R_s).

  First, we show that C_q must be in the interval (W_q, R_q). By Lemma 5, there must be a writeback of q's cache entry that is initiated and completed in the interval (W_q, R_q). Since I_q is the first writeback initiation of q's cache entry after W_q and C_q is the corresponding writeback completion, it must be that C_q is in (W_q, R_q).

  Next, we show that C_q must be in the interval (W_s^*, R_s). By Lemma 5, there must be a writeback of s's cache entry that is initiated and completed in the interval (W_s^*, R_s). Since C_q is the completion of the last writeback to ℓ before Rd_p^M, it must be that C_q is in (W_s^*, R_s).

  But since R_q precedes W_s^*, C_q cannot be in both (W_q, R_q) and (W_s^*, R_s).

Finally, we return to ρ_{mm}Ṡince W_s^* does not exist in any case, readOK?(w, p) at Rd_p^M, so p reads v. □

Lemma 8 *In ρ_{cp}, let Rd_p^C be a move at which a* ProcAgent *p performs a clean read of a* Value *v. Then in ρ_{mm}, p also reads v at Rd_p^C.*

Proof. First, consider ρ_{cp}. At Rd_p^C, p.cacheVal $= v$. For the read to be clean, it must be the case that p.cacheValid? and not p.cacheDirty?. Initially, not p.cacheValid?, so this is only possible if one of the following operations has occurred earlier:

- A miss read by p (which updates p.cacheValid? to true and p.cacheDirty? to false).

- A writeback of p's cache entry (which is only initiated if p.cacheValid?, and which updates p.cacheDirty? to false) resulting from a release, that therefore does not eject the entry (leaving p.cacheValid? unchanged).

Let MI_p be the move of the last such operation (either a miss read or a writeback initiation) before Rd_p^C. Then in the interval (MI_p, Rd_p^C):

- any read by p leaves p.cacheVal unchanged, since p.cacheValid?;
- p does not write, since not p.cacheDirty? at Rd_p^C. A write by p in the interval (MI_p, Rd_p^C) would update p.cacheDirty? to true, and there is no writeback of p's cache entry in (MI_p, Rd_p^C) to update p.cacheDirty? to false (since MI_p is the last such writeback).

Thus p.cacheVal is unchanged in the interval (MI_p, Rd_p^C).
MI_p is either a miss read move or a writeback initiation move. We consider each possibility in turn.

- If MI_p is a miss read, then we give MI_p a more descriptive name: let $Rd_p^M = MI_p$. In ρ_{cp}, p reads v at Rd_p^M.

  Now consider ρ_{mm}. We show that in this run, p also reads v at Rd_p^C. By Lemma 7, p reads v at Rd_p^M. Hence there is a WriteEvent w such that w.val $= v$ and readOK?(w, p) at Rd_p^M. Since readOK?(w, p) at Rd_p^M, one of the following must be true:

  - At Rd_p^M, $w \not\prec p$.latestEvent. If not readOK?(w, p) at Rd_p^C, then $w \prec p$.latestEvent at Rd_p^C. By Lemma 4, this is true only if p acquires in the interval (Rd_p^M, Rd_p^C).
  - At Rd_p^M, $w \preceq p$.latestEvent, but there is no write event w^* for which $w \prec w^* \preceq p$.latestEvent. If not readOK?(w, p) at Rd_p^C, then there is a WriteEvent w^* such that $w \prec w^* \prec p$.latestEvent at Rd_p^C. Since p does not write in the interval (Rd_p^M, Rd_p^C), w^* cannot be a p-write. So by Lemma 4, p must acquire in the interval (Rd_p^M, Rd_p^C).

  Returning to ρ_{cp} we show that p does not acquire in (Rd_p^M, Rd_p^C). Assume to the contrary that p does acquire at A_p^* in (Rd_p^M, Rd_p^C). Then at A_p^*, p.cacheValid? is updated to false, but p.cacheValid? at Rd_p^C, so there must be a write or miss read by p in (A_p^*, Rd_p^C). But as shown earlier, this is not the case.

  Finally, we return to ρ_{mm}Ṡince p does not acquire in the interval (Rd_p^M, Rd_p^C), readOK?(w, p) at Rd_p^C, so p reads v.

- If MI_p is a writeback initiation move, then let $I_p = MI_p$. At I_p, p.cacheVal $= v$.

  Let W_p be the last write operation by p (updating p.cacheDirty? to true) before I_p. Since I_p is a writeback initiation move, p.cacheDirty? at I_p. Then in the interval (W_p, I_p), p.cacheDirty?, so by Claim 2, p.cacheValid?; hence any read by p leaves p.cacheVal unchanged. Thus the value written at W_p is v.

  Now consider ρ_{mm}. We show that in this run, p also reads v at Rd_p^C. Let w_p be the write event issued at W_p; then w_p.val $= v$. At Rd_p^C, not readOK?(w_p, p) only if there is a write event $w^* \succ w_p$ that precedes p.latestEvent.

Returning to ρ_{cp}, we show that w^* cannot exist. Assume to the contrary that such a w^* exists; let W^* be the move at which w^* is issued. Since p does not write in the interval (W_p, Rd_p^C), w^* is not a p-write event. There are two cases:

- W^* is in the interval (W_p, I_p). By Lemma 4, such a W^* exists only if p releases in the interval (W_p, W^*).

  Returning to ρ_{cp} we show that p cannot release in the interval (W_p, W^*). First, note that in this interval, no writeback of p's cache entry is initiated, since it would update p.cacheDirty? to false, and there is no write in the interval to update it back to true.

  Assume to the contrary that p does release at R_p^* in (W_p, I_p). By Lemma 5, in (W_p, R_p^*) there must be a writeback of p's cache entry. But as shown above, there is no such writeback initiated.
- W^* is in the interval (I_p, Rd_p^C). By Lemma 4, such a w^* exists only if p acquires in the interval (W^*, Rd_p^C).

  Returning to ρ_{cp} we show that p does not acquire in (I_p, Rd_p^C). Assume to the contrary that p does acquire at A_p^* in (Rd_p^M, Rd_p^C). Then at A_p^*, p.cacheValid? is updated to false, but p.cacheValid? at Rd_p^C, so there must be a write or miss read by p in (A_p^*, Rd_p^C). But as shown earlier, this is not the case.

Finally, we return to ρ_{mm}Ṡince w^* does not exist, readOK?(w_p, p), so p reads v at Rd_p^C.

□

Theorem: LC_{cp} **obeys** LC_{mm}

Theorem 1 *Let Rd_p be a move of ρ_{cp} at which a* ProcAgent *p reads a* Value *v. Then at Rd_p in ρ_{mm}, p also reads v.*

Proof. Immediate from Lemmata 6–8. □

6 LC_{cp} is strictly stronger than LC_{mm}

We now show that LC_{cp} disallows certain behavior allowed by LC_{mm}. In particular, we give an execution of LC_{mm} in which a particular value is read; we then show that this value cannot be read in any equivalent run of LC_{cp}.

Consider a run ρ_{mm} of LC_{mm} with the following properties. In ρ_{mm}, two distinct processor agents p and q operate on a common location ℓ, and no other processor agents operate on ℓ. (Other processor agents may perform operations on locations other than ℓ.) The operations of p and q occur in the following sequence:

A_p: Acquire by p.

W_p: Write by p, that writes the value 1.

R_p: Release by p.

W_q: Write by q, that writes the value 2.

A_q: Acquire by q.

Rd_q: Read by q.

First we show that at Rd_q, 1 is a readable value, according to LC_{mm}. (Note that 2 is also a readable value.)

Lemma 9 *In ρ_{mm}, q reads the* Value 1 *at Rd_q.*

Proof. Let w_p be the WriteEvent generated at W_p. We show that q reads w_p at Rd_q. At Rd_q, not readOK?(w_p, q) only if there is a WriteEvent w^* such that $w_p \prec w^* \prec q$.latestEvent. Assume that there is such a w^*. Since the only write to ℓ after W_p is W_q, w^* must be generated at W_q. But since q does not acquire in the interval (W_p, W_q), not $w_p \prec q$.latestEvent at W_q by Lemma 4, so not $w_p \prec w^*$. Hence there is no such w^*, so readOK?(w_p, q) at Rd_q and therefore q reads the Value 1. □

Next, we show that in any equivalent run of LC_{cp}, 2 is the value read at Rd_q. (Note that in any run of LC_{cp}, a processor agent reads only a single value at a time, so q does *not* read 1 at Rd_q.)

Lemma 10 *Let ρ_{cp} be any run of LC_{cp} equivalent to ρ_{mm}. Then in ρ_{cp}, q reads the* Value 2 *at Rd_q.*

Proof. At W_q, q.cacheValue is updated to 2, q.cacheValid? is updated to true, and q.cacheDirty? is updated to true. There are two possibilities:

- A writeback of q's cache entry is initiated at I_q in the interval (W_q, Rd_q). Since q does not release in (W_q, Rd_q), I_q must be an ejection of q's cache entry (due to a read or write by some ProcAgent other than p or q). Since q does not read or write in (W_q, I_q), q.cacheVal is unchanged in the interval, so q.cacheVal = 2 at I_q. At I_q, a WritebackAgent wb_q is generated, wb_q.val is updated to q.cacheVal = 2, wb_q.active? is updated to true, q.latestWB is updated to wb_q, and q.cacheDirty? is updated to false.

  There is no writeback of q's cache entry initiated in the interval (W_q, I_q); any such writeback would update q.cacheDirty? to false, but q.cacheDirty? at I_q and there is no write by q in (W_q, I_q) to update q.cacheDirty? back to true. Therefore q.latestWB = wb_q at Rd_q. Furthermore, by Lemma 5, a writeback of p's cache entry is initiated and completed in the interval (W_p, R_p), so not p.cacheDirty? at R_p. Since p does not read or write in (R_p, Rd_q), p.cacheDirty? is unchanged in the interval, so there is no writeback of p's cache entry.

  There are two possibilities:

  - wb_q completes the writeback at C_q in (I_q, Rd_q). At C_q, ℓ.MMVal is updated to 2, and wb_q.active? is updated to false. Since there is no writeback by p or q initiated in (I_q, Rd_q), there is no writeback completion in (C_q, Rd_q), so ℓ.MMVal is unchanged in this interval. Thus at Rd_q, not q.cacheValid?, and not wb_q.active?, so q reads the Value ℓ.MMVal = 2.
  - wb_q does not complete the writeback in (I_q, Rd_q). Then at Rd_q, wb_q.active? and q.latestWB = wb_q, so q reads the Value wb_q.val = 2.

- No writeback of q's cache entry is initiated in (W_q, Rd_q). Then q.cacheDirty? is unchanged in (W_q, Rd_q), so q.cacheDirty? at Rd_q. By Claim 2, q.cacheValid? at Rd_q, so q reads the Value q.cacheVal at Rd_q. Since q performs no read or write in (W_q, Rd_q), q.cacheVal is unchanged in the interval, so q.cacheVal = 2 at Rd_q.

Theorem: LC_{cp} is strictly stronger than LC_{mm}

Theorem 2 *There exists a run ρ_{mm} of LC_{mm} in which a read operation Rd returns a* Value *v that cannot be returned by the same read operation in any equivalent run of LC_{cp}.*

Proof. Consider the run ρ_{mm} described earlier in this section. By Lemma 9, q reads the Value 1 at Rd_q in ρ_{mm}. But by Lemma 10, q reads the Value 2 (and therefore does not read 1) at Rd_q in any run of LC_{cp} equivalent to ρ_{mm}. □

7 Conclusion

In this paper, we have presented formal specifications for the LC memory model and for the LC cache protocol. These specifications, contrary to the descriptions presented in [9] or [10], have been expressed rigorously using the formal ASM specification language. Using these formal specifications and the notions of sequential and distributed runs, we have then been able to show that the protocol indeed satisfies the model. In other words, we have shown that, using the LC protocol, any value returned by a read operation is a value legal according to the LC memory model.

In its current state, the ASM methodology has only a rudimentary notion of automated verification compared to process algebra or other techniques. For example, using the ASM approach, it is not currently possible to formally show the equivalence of the two models LC_{mm} and LC_{cp} by finding an appropriate bisimulation relation between them, as is possible with process algebra descriptions [18]. An interesting area of further study would thus be to explore the use of techniques such as model checking to automate portions of the proof. Model checking of ASM specification is an active area of research [4].

Another interesting question to explore would be to determine whether the protocol is as strong, or weaker, than the memory model, that is, are there some events allowed by the memory model which are precluded by the protocol? Finally, another interesting area of future research would be to express other weak memory models using the same approach and see how these models, and their associated protocols, differ/compare with the LC memory model.

A Abstract state machines

In this appendix, we provide a brief introduction to ASM; for further description and discussion, consult the standard ASM language guides [12, 13]. For illustrative purposes, we include a running example: the well known algorithm of Euclid [7, 16] for finding the greatest common divisor of two natural numbers.

A.1 Vocabulary

As in classical (more exactly, first-order) logic, a *vocabulary* is a collection of function and relation names, with each name assigned a natural number as its arity. Deviating slightly from classical

tradition, we view relation names as special function names (called predicates). We consider only vocabularies that contain (at least) the nullary (0-ary) predicates true and false, the nullary function name undef, the equality predicate and the standard Boolean connectives treated as predicates. Since these names appear in each vocabulary, they are usually omitted when a vocabulary is described. In fact, the vocabulary of an ASM is often given implicitly, as the collection of those function names that occur in the program of the ASM.

For our implementation of Euclid's algorithm we define a vocabulary Υ_0 which contains (in addition to the obligatory names) the nullary function names mode, Compute, Final, 0, 1, left, right, output, the unary function name Nat, and a binary function name mod.

We use these function names in the following way. The ASM starts in Compute mode and continues executing until it is in Final mode; the current mode of the ASM is given by mode. Euclid's algorithm requires two variables for storing natural numbers; initially, these variables contain the two numbers for which the user wishes to compute the greatest common divisor. These variables are represented by left and right. The final output of the algorithm is given by output. Finally, we use Nat to represent the set of natural numbers, 0 and 1 to represent the natural numbers 0 and 1, and mod to represent the remainder function defined over the natural numbers.

A.2 States and updates

For those familiar with classical logic, we can say that a *state* S of a vocabulary Υ is a structure of vocabulary Υ where the interpretations of the obligatory names satisfy the standard constraints; in particular, the elements (which interpret) true and false are distinct, and any Boolean connective has value false if at least one of its arguments is non-Boolean. In addition, the only possible values of (the interpretation of) any predicate are true and false.

In other words, S consists of a nonempty set X, called the *basic set* of S, and an interpretation of each function name in Υ over X. A r-ary function name is interpreted as a r-ary function from X^r to X. Intuitively, (the interpretation of) undef represents an undefined value and is used to represent partial functions. While in principle every basic function f (*i.e.*, the interpretation of every function name "f") in the state is total, we usually define the domain for f and then set its value to undef for every tuple outside its domain. We think of a basic relation R (*i.e.*, the interpretation of predicate R) as the set of tuples where R "is true" (*i.e.*, takes the value true). This explains why the default value of Boolean connectives is false rather than undef. Some basic unary relations may be called *universes* and used to define domains of basic functions.

To continue our example, we describe a class K_0 of states S of vocabulary Υ_0 mentioned above. The members of K_0 are the valid initial states of our implementation of Euclid's algorithm. The basic set of S consists of the natural numbers together with three elements (which interpret) true, false and undef. The function 0 means (*i.e.*, is interpreted by) the number zero, the function 1 means one, and the function mod is (interpreted by) the standard arithmetic function (*e.g.*, seven mod three equals one). The functions Compute and Final mean zero and one respectively. The function mode means zero. The functions left and right are natural numbers. The function Nat represents the universe of natural numbers; it is (interpreted by) the predicate which holds for all and only the elements of the basic set that are natural numbers.

We commonly associate each function with a *profile*, indicating the intended domain and range of each function. For instance, a function with profile Nat $\rightarrow$ Nat maps elements of Nat to elements of Nat. Function profiles are not types; rather, they are an informal means to illustrate how each function is used. We represent the vocabulary Υ_0 in tabular form, as shown in Figure 15.

Function	Profile/Description
0, 1	Nat The natural numbers zero and one.
Compute, Final	Nat Execution modes.
mode	Nat Current execution mode.
left, right	Nat Working values.
output	Nat Output value.
x mod y	Nat $\times$ Nat $\rightarrow$ Nat The arithmetic modulo operator.

Figure 15: Functions for sequential ASM.

An *update* is an atomic state change, that is, a change in the interpretation of a single function name for a single tuple of arguments. We formalize the notion of an update as follows. A *location* in a state S is a pair $\ell = (f, \bar{x})$, where f is an r-ary function name in the vocabulary of S and $\bar{x}$ is an r-tuple of elements (of the basic set) of S. Then an update of S is a pair (ℓ, y), where ℓ is a location of S and y is an element of S. To *fire* an update (ℓ, y) at a state S, bind y to the location ℓ (*i.e.*, redefine S to map ℓ to y).

To avoid expanding the basic set of the current state of a program when modeling the dynamic creation of elements, it may be convenient to endow states with a *reserve*. Formally, an element a is in the reserve if (1) every basic relation returns false when given a as one of its arguments; (2) every other basic function returns undef when given a as one of its arguments; (3) no basic function returns the element. When a *new* element is needed, it is simply taken from the reserve.

A.3 Terms and transition rules

Terms of a given vocabulary Υ are defined inductively. For succinctness, we define the following within the same inductive definition: *i*) the class of terms; *ii*) the (sub)class of *Boolean terms*; *iii*) the function [ ] that gives the value $[t]_S$ of a term t given an Υ-state S and a mapping Val_S of the variables of t to elements of S.

- Any variable v is a term.

  $[v]_S$ is given by the mapping Val_S, i.e., $Val_S(v)$.

- If f is an r-ary function name and $t_1, \ldots, t_r$ are terms, then $f(t_1, \ldots, t_r)$ is a term.

  If f is a predicate, then $f(t_1, \ldots, t_r)$ is a Boolean term.

  $[f(t_1, \ldots, t_r)]_S$ is defined as $f^S([t_1]_S, \ldots, [t_r]_S)$, where f^S is the interpretation of f in S.

- If $g(v)$ and $t(v)$ are Boolean terms where v is a variable, then $(\exists v : g(v))t(v)$ and $(\forall v : g(v))t(v)$ are Boolean terms.

  The values of the quantified terms are defined in the expected way, *e.g.*, $[(\exists v : g(v))t(v)]_S$ is true if there exists a value v such that $[g(v)]_S$ and $[t(v)]_S$ are both true.

  We use $(\forall v : U : g(v))t(v)$, where U is a universe name, to abbreviate $(\forall v\text{: } U(v) \text{ and } g(v))\ t(v)$.

One may use infix notation for binary functions. For example, left mod right is a term in the vocabulary Υ_0. Postfix notation can also be used, for example, we use t.undef?, where t is a term, as an abbreviation for $t =$ undef.

An ASM performs state updates through *transition rules*. For a given state S, a rule gives rise to a set of updates as follows.

- If R is an *update instruction*, of the form $f(t_1, \ldots, t_r) := t_0$, where f is an r-ary function name and each t_i is a term, then the update set of R at S contains a single update (ℓ, y), where $y = [t_0]_S$ and $\ell = (f, ([t_1]_S, \ldots, [t_r]_S))$. To execute R at S, set $f^S([t_1]_S, \ldots, [t_r]_S)$, where f^S is the interpretation of f in S, to $[t_0]_S$.

- If R is a *conditional rule*, of the form

  if g_0 **then** R_0
  $\vdots$
  elseif g_n **then** R_n
  endif

  where $g_0 \ldots g_n$ (the *guards*) are Boolean terms and $R_0 \ldots R_n$ are rules, then the update set of R at S is the update set of R_i at S, where i is the minimum value for which g_i evaluates to true. If all g_i evaluate to false, then the update set of R at S is empty. To execute R at S, execute R_i if it exists; otherwise do nothing.

  If there is a "default" rule R_n to execute when all guards $g_0 \ldots g_{n-1}$ fail, we use the following abbreviation:

if g_0 **then** R_0	abbreviates	**if** g_0 **then** R_0
elseif g_1 **then** R_1		**elseif** g_1 **then** R_1
$\vdots$		$\vdots$
elseif g_{n-1} **then** R_{n-1}		**elseif** g_{n-1} **then** R_{n-1}
else R_n		**elseif** true **then** R_n

  In a situation where different actions are taken based on the value of a term t_0, we use the following abbreviation:

case t_0 **of**	abbreviates	
t_1: R_1		**if** $t_0 = t_1$ **then** R_1 **endif**
$\vdots$		$\vdots$
t_n: R_n		**if** $t_0 = t_n$ **then** R_n **endif**
endcase		

  Finally, for brevity we often omit the "end" keywords such as **endif**, using indentation to indicate the intended level of nesting.

- If R is an *import rule*, of the form

  import v
  R_0
  endimport

  where v is a variable name and R_0 is a rule, then the update set of R at S is the update set of R_0 at $S(v \to a)$, where a is a reserve element. To execute R at S, execute R_0 at $S(v \to a)$. When an imported element is to be added to a universe U, we use the following abbreviation:

  extend U **with** v abbreviates **import** v
  R_0 $U(v) :=$ true
  endextend R_0
  endimport

- If R is a *block rule*, of the form

  do-inparallel
  R_0
  $\vdots$
  R_n
  enddo

  where $R_0 \ldots R_n$ are transition rules, then the update set of R at S is the union of the update sets of each R_i. To execute R at S, execute all R_i simultaneously. We often abbreviate such a rule as $R_0 \ldots R_n$, omitting the keywords.

- If R is a *for-all rule*, of the form

  do-forall v: g
  R_0
  enddo

  where v is a variable name, g is a Boolean term and R_0 is a rule, then the update set of R is the union of the update sets of R_0 at all $S(v \to a)$, where a is any element of S for which g evaluates to true at $S(v \to a)$. To execute R at S, execute R_0 at $S(v \to a)$ for all such a. We also use the following abbreviation:

  do-forall v: U: g abbreviates **do-forall** v: $U(v)$ and g
  R_0 R_0
  enddo **enddo**

For certain rules (more specifically, certain rules with block or for-all components), execution may result in updates of the same location to different values. Obviously, such updates cannot be performed simultaneously. If an update set contains updates (l, x) and (l, y), where $x \neq y$, then it is *inconsistent*; otherwise it is *consistent*. To fire a consistent update set at a given state, fire all its members simultaneously. To fire an inconsistent update set, do nothing.

To reduce the size of our rules and avoid repetition, we often give names to certain terms and rules which appear within (larger) rules. We then use the name of a term/rule rather than the term/rule itself when it appears within a rule. We also use parameterized terms and rules. To instantiate a parameterized term or rule, give its name with each parameter name replaced by

```
rule Compute GCD:
if mode = Compute then
    if right = 0 then
        output := left
        mode := Final
    else
        left := right
        right := left mod right
```

Figure 16: Rule for sequential ASM.

an ASM term/rule. The result of the instantiation is achieved by replacing each instance of a parameter name with the corresponding term/rule.

For our implementation of Euclid's algorithm, we define a rule named *Compute GCD*. The rule is given in Figure 16. Given appropriate initial values associated with left and right and an initial mode defined as Compute, *Compute GCD* finds the greatest common divisor of left and right by repetitively performing the state transitions specified by the update rule. At each repetition, the following updates are performed *in parallel*: transferring the right value to the left variable, and storing the remainder of left and right to the right variable.[18] Once the right variable attains the value 0, the ASM sets its final output to the current value of left and then enter its Final state.

A *program* is a rule without free variables. A *sequential ASM* $\mathcal{A}$ consists of a vocabulary Υ, a class K of Υ-states (the *initial states* of $\mathcal{A}$), and a program P of vocabulary Υ. For example, the vocabulary Υ_0, the class K_0 of states, and the program *Compute GCD* constitute a sequential ASM. A *pure run* of a program P is a sequence $S_0, S_1, \ldots$ of states appropriate to P such that each S_{n+1} is obtained from S_n by firing (the update set of) P at S_n.

We call such a run a *pure* run because it is not affected by the environment; all state changes are caused by program execution. In general, however, runs of certain programs may be affected by the environment; for example, a program may respond to user input during its execution. Functions that manifest the environment are called *external* or *monitored* functions. An external function acts as a (dynamic) oracle. It need not be consistent over multiple steps; that is, it may give different results at different steps, even when all inputs are the same. However, the oracle is consistent during a single step; if an external function is evaluated multiple times in the same step, it returns the same value each time.

We call non-external basic functions *internal*. If S is an appropriate state for a program P, let S^{int} be the reduct of S to the internal vocabulary. A *run* of P is a sequence of states $S_0, S_1, \ldots$ such that (1) every non-final S_n is an appropriate state for P and the final state (if any) is a state of the internal vocabulary of P; and (2) every S^{int}_{n+1} is obtained from S_n by firing P at S_n.

[18] It should be stressed that the order of the rules in a block is arbitrary, since the corresponding updates happen in parallel. Thus in the example, the updates "left := right" and "right := left mod right" could appear in the opposite order, with no effect on behavior.

A.4 Agents and distributed runs

In a distributed computation, multiple agents act concurrently, each according to its own program. Distributed ASMs are an extension of sequential ASMs in which some elements of the basic set are identified as agents and are assigned programs. A special function Self is used for agent self-reference. The notion of a run is enriched to reflect the concurrency of agents' executions.

A *distributed ASM* $\mathcal{A}$ consists of

- A finite indexed set of programs π_ν, called *modules*. Each module name ν is a static nullary function name.
- A vocabulary Υ which includes every function name that appears in any π_ν, except for the function name Self. In addition, Υ contains a unary function name Mod that, given an agent, returns its associated module name; Υ also contains a unary function name Prog that, also given an agent, returns its associated module.
- A collection of Υ-states, the *initial states* of $\mathcal{A}$, satisfying the following conditions:
  - Different module names are interpreted as different elements.
  - For every module name ν, there are only finitely many elements a such that $\mathsf{Mod}(a) = \nu$.

An Υ-state is a *(global) state* of $\mathcal{A}$ if it satisfies the two conditions imposed on initial states. An element a is an *agent* at S if there is a module name ν such that $\mathsf{Mod}(a) = \nu$ at S. The program π_ν is the program $\mathsf{Prog}(a)$ of agent a.

We give an example of a distributed ASM by modifying our sequential ASM of the previous section. Instead of a single pair of values as input, we take a set of pairs. In addition to computing the GCD of each pair, the program determines whether any of the pairs are relatively prime; this is true if the right value of any pair ever reaches the value 1. We implement this as a distributed ASM, in which a set of agents (members of the universe GCD), computes the greatest common divisor of each pair.

We define the vocabulary Υ_1 by making the following changes to Υ_0, as shown in Figure 17. Note that the function primePair? does not have an agent parameter; it serves as a global flag shared by all agents.

The class K_1 of initial states is described as follows. In any initial state, the function primePair? is false, and for each GCD agent a, $\mathsf{mode}(a) = \mathsf{Compute}$, $\mathsf{left}(a)$ and $\mathsf{right}(a)$ are natural numbers, and $\mathsf{output}(a) = \mathsf{undef}$. The single module of the ASM is given in Figure 18.

Distributed ASMs require a more elaborate notion of run. Observers of a distributed execution may differ in their perceptions of the execution; in particular, their histories of the execution may differ in the relative order of concurrent actions. A run of a distributed ASM can be seen as the common part of all observers' histories.

For every agent a and state σ, $\mathsf{View}(a, \sigma)$ (the *local state* of a) is the reduct of σ to the functions appearing in $\mathsf{Prog}(a)$, extended by interpreting the function name Self as a. To fire a at state σ, execute $\mathsf{Prog}(a)$ at state $\mathsf{View}(a, \sigma)$. A *run* ρ of a distributed ASM $\mathcal{A}$ is a triple (μ, α, σ), where

- μ, the *moves* of ρ, is a poset (partially ordered set) where for every $M \in \mu$, $\{N : N \leq M\}$ is finite. If μ is totally ordered, we say that ρ is *sequential*.

Function	**Profile/Description**
mode(a)	GCDAgent → Nat Current execution mode of agent a.
left(a), right(a)	GCDAgent → Nat Working values for agent a.
output(a)	GCDAgent → Nat Output value for agent a.
primePair?	Boolean Is there a relatively prime pair?

Figure 17: Functions of distributed ASM.

```
module GCD:
if mode(Self) = Compute then
    if right(Self) = 0 then
        output(Self) := left(Self)
        mode(Self) := Final
    else
        if right(Self) = 1 then primePair? := true
        left(Self) := right(Self)
        right(Self) := left(Self) mod right(Self)
```

Figure 18: Module for distributed ASM.

- α is a function mapping moves to agents so that for every agent a, every nonempty set $\{M : \alpha(M) = a\}$ is linearly ordered. (For each move M, $\alpha(M)$ is the agent performing M. The condition on α ensures that every agent's execution is sequential.)

- σ is a function mapping finite initial segments of μ to states of $\mathcal{A}$. The state $\sigma(\emptyset)$ is an initial state. (The state $\sigma(\nu)$, where ν is a finite initial segment of μ, is the result of performing all moves of ν in the order given by μ. A state σ of $\mathcal{A}$ is *reachable* in a run ρ if it is in the range of σ.)

- Coherence condition: If X is a maximal element in a finite initial segment ν of μ and $\nu' = \nu - \{X\}$, then $\alpha(X)$ is an agent in $\sigma(\nu')$, X is a move of $\alpha(X)$ and $\sigma^{int}(\nu)$ is obtained from $\sigma(\nu')$ by performing X at $\sigma(\nu')$.

Certain pairs of moves are seen by all observers as executing in a particular order. These moves are ordered according to the poset μ. Other pairs of moves may be seen in different orders by different observers. These moves are unordered. Note that the predecessors of a given move may not be totally ordered. For instance, imagine a run in which agent a and agent b make moves that are unordered with respect to each other. It is possible that agent c makes a move that is ordered after both a and b. Intuitively, observers may disagree about the relative order of a's move and b's move, but all agree that of the three moves, c's is the last.

Note that the state function σ maps finite initial segments of μ to states. This, along with the coherence condition, implies that for each finite initial segment of moves, there is a single state σ such that all observers who witness all (and only) the moves in that finite initial segment agree on σ as the current global state. In our example, all observers who have witnessed a's move and b's move must have a common notion of what the current global state is.

It would be useful to be able to speak of "the state at move M", but in a distributed run, there may not be a unique state associated with M. Nevertheless, we can speak of a property holding at a move M, meaning that for *every* state associated with M, the property holds. We formalize this as follows.

Definition *Let M be a move of μ.*

- *For any finite initial segment ν of μ that has M has a maximal element,* $\Sigma(\nu)$ is a state at M.
- *Let ϕ be a proposition that holds for every state at M. Then ϕ* holds at *M.*
- *Let ϕ be a proposition that holds at every successor of M. Then ϕ holds* forever after *M.*

There is one slight complication involving monitored functions. The coherence condition on runs does not require monitored functions to be interpreted identically at the various states associated with a move. This makes it impossible to state that a property involving a monitored function holds at a given move. The freedom to interpret monitored functions differently at different states of a move does not give us anything, so we place the following restriction on runs.

- Let ν_1 and ν_2 be finite initial segments of μ, and let X be a maximal element in both ν_1 and ν_2. Then $\sigma^{ext}(\nu_1) = \sigma^{ext}(\nu_2)$.

It is often more convenient to reason about a sequential run than a distributed one. A *linearization* of a distributed run is a sequential run that preserves the partial order on moves in the distributed run. More formally, a linearization of a poset π is a linearly ordered set π' with the same elements as π such that if $X < Y$ in π then $X < Y$ in π'. A run ρ' is a linearization of a run ρ if the move poset of ρ' is a linearization of that of ρ, the agent function of ρ' is that of ρ, and the state function of ρ' is a restriction of that of ρ. The following corollary in the ASM Lipari guide [12] relates a distributed run to its linearizations:

Corollary *A proposition holds in every reachable state of a run ρ if and only if it holds in every reachable state of every linearization of ρ.*

This implies that we can reason about a distributed run by considering all its possible linearizations instead of the distributed run itself.

References

[1] S.V. Adve and K. Gharachorloo (1995). Shared memory consistency models: a tutorial. Research Report 95/7, Digital Western Research Laboratory.

[2] B. Bershad, M. Zekauskas and W. Sawdon (1993). The Midway distributed shared memory system. in *Proceedings of the IEEE COMPCON*.

[3] R.D. Blumofe, M. Frigo, C.F. Joerg, C.E. Leiserson and K.H. Randall (1996). An analysis of DAG-consistent distributed shared-memory algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 297–308.

[4] G. Del Castillo and K. Winter (2000). Model checking support for the ASM high-level language. In *Proceedings of TACAS 2000*, S. Graf and M. Schwartzbach (editors), LNCS 1785, Springer-Verlag, 331-346.

[5] S. Cook and R. Reckhow (1973). Time bounded random access machines. *Journal of Computer and System Sciences* 7, 354–375.

[6] D.E. Culler and J.P. Singh, with A. Gupta (1999). *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann.

[7] Euclid (1956). *The Thirteen Books of Euclid's Elements, Volume II: Books III–IX*. T. Heath (editor and translator). Dover.

[8] G.R. Gao. Personal communication.

[9] G.R. Gao and V. Sarkar (1994). Location consistency: Stepping beyond the barriers of memory coherence and serializability. ACAPS Technical Memo 78, School of Computer Science, McGill University.

[10] G.R. Gao and V. Sarkar. Location consistency — A new memory model and cache consistency protocol. *IEEE Trans. on Comp.*, 49(8):798–813, August 2000.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy (1990). Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 15–26. Also in *Computer Architecture News* 18(2).

[12] Y. Gurevich (1995). Evolving Algebras 1993: Lipari guide. In E. Börger (editor), *Specification and Validation Methods*, Oxford University Press, 9–36.

[13] Y. Gurevich (1997). May 1997 draft of the ASM guide. Available at `http://www.eecs.umich.edu/gasm/`.

[14] Y. Gurevich and D. Rosenzweig. Partially ordered runs: A case study. In *Abstract State Machines: Theory and Applications*, pages 131–150. Springer-Verlag, LNCS-1912, 2000.

[15] P. Keleher, A.L. Cox and W. Zwaenepoel (1992). Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 13–21. Also in *Computer Architecture News* 20(2).

[16] D. Knuth (1973). *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley.

[17] L. Lamport (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* C-28(9): 690–691.

[18] R. Milner (1989). *Communication and concurrency*. Prentice-Hall.

[19] D.L. Parnas and J. Madey (1995). Functional documentation for computer systems. *Science of Computer Programming* 25(1): 41–61.

[20] D. Perrin (1990). Finite automata. In *Handbook of Theoretical Computer Science*, ed. J. van Leeuwen, Elsevier, 1–57.

[21] A. Turing (1936). On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society* 2(42), 230–265.