A Unified Formal Specification and Analysis of the new Java Memory Models

By
Varsha Awhad

A THESIS
Submitted in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE

MICHIGAN TECHNOLOGICAL UNIVERSITY
2002

This thesis, "A Unified Formal Specification and Analysis of the new Java Memory Models ", is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

DEPARTMENT Computer Science

Signatures:

Thesis Advisor: _____
Dr. Charles Wallace

Department Chair: _____
Dr. Linda Ott

Date: _____

Acknowledgments

This thesis has reached its completion because of the guidance and support of several people. I would like to thank my advisor Dr. Charles Wallace, for his consistent encouragement and excellent guidance. It has been a privilege working under him. I would also like to thank Guy Tremblay, for his comments on an earlier draft of this paper, and José Nelson Amaral, for his thoughts on Location Consistency. I would like to thank our department chair Dr. Linda Ott, and my committee members Dr. Steve Seidel and Dr. Allan Struthers, for their insights, time and understanding support.

I would like thank my parents, without whom this higher education would not have been possible. Last but not the least, I would like to thank my friend Pratik Chandan for his moral support. He has been with me at every step.

Abstract

In a multithreaded program running on a multiprocessor platform, different processors may observe operations in different orders. This may lead to surprising results not anticipated by the programmer. The problem is exacerbated by common compiler and hardware optimization techniques. A memory (consistency) model provides a contract between the system designer and the software designer that constrains the order in which operations are observed. Every memory model strikes some balance between strictness (simplifying program behavior) and laxness (permitting greater optimization).

With its emphasis on cross-platform compatibility, the Java programming language needs a memory model that is satisfactory to language users and implementors. Everyone in the Java community must be able to understand the Java memory model and its ramifications. The description of the original Java memory model suffered from ambiguity and opaqueness, and attempts to interpret it revealed serious deficiencies. Two memory models have been proposed as replacements. Unfortunately, these two new models are described at different levels of abstraction and are represented in different formats, making it difficult to compare them.

In the work presented here we formalize these models and develop a unified representation of them. We do so using Abstract State Machines, a formal operational semantics methodology. Using the mathematical models we develop we compare the two Java Memory Models to a modified form of the Location Consistency model which we propose as a model for Java. In doing so we takesteps towards developing a generic framework that can be applied to any model for the purpose of study and comparison.

Contents

1 Introduction

As parallelism becomes ever more prevalent in the computing world, the *shared memory* paradigm is emerging as a popular approach to building parallel systems and applications. A *shared memory multiprocessor machine* is characterized by a collection of processors that exchange information with one another through a *global address space* [1, 7]. In such machines, memory may be partitioned in many ways. For instance, in bus-based shared memory, processors access a separate monolithic main memory area over a bus; in distributed shared memory, each processor may have local access to some portion of the shared memory, but remote memory accesses require communication via a network. The advantage of shared memory is that programmers writing applications for such machines are shielded from details about the physical distribution of memory. Each processor views the same global address space and accesses remote memory locations through the standard read and write instructions found in uniprocessor machines, rather than through special message-passing operations [14].

The behavior of a program running on a shared memory machine depends on the order in which memory operations are made visible to processors. Due to the physical distribution of memory, operations cannot be made visible instantaneously to all processors, so there is no simple notion of a single order of operations. Indeed, many common optimization techniques may complicate the ordering even further:

- Shared memory machines have various buffers where data written by a processor can be stored before they are shared with other processors. Freshly written values may remain in write buffers until it is convenient to propagate them to other processors. Processors may choose to read outdated values from their local caches to avoid the cost of remote reads.
- Compilers may permute program instructions into an order more advantageous from a performance perspective. Similarly, processors may execute instructions out of program order. Typically, such reorderings are chosen carefully so as to be imperceptible in a uniprocessor context, but they cannot be hidden from other processors operating concurrently on a shared address space.

In a totally unregulated shared memory machine, programmers would find it impossible to predict the behavior of their applications. Optimizations at the hardware or compiler level can have consequences for even simple programs. Consider the fragments of Java code in Fig. 1. A programmer accustomed to the simple uniprocessor model might be surprised to find Thread T_1 reading the value 1 written by Thread T_2, or likewise T_2 reading the value 1 written by T_1, but this is of course an inevitable consequence of shared

P refers to the same object in both code fragments;
Initially, `P.x == 0` and `P.y == 0`

Thread T_1	Thread T_2
`P.x = 1;`	`P.y = 1;`
`result1 = P.y;`	`result2 = P.x;`

Figure 1: A simple parallel program.

memory. On the other hand, the programmer would be even more surprised to find both T_1 and T_2 reading the value 0. If T_1 reads 0, it seems intuitive that at the time of its read, (1) T_1's write to `P.x`, appearing as it does before T_1's read in program order, must have completed; (2) T_2's write of 1 to `P.y` must not have completed, since T_1 read the value 0 for `P.y`; (3) T_2's read of `P.x`, since it follows T_2's write in program order, must not have completed. From this reasoning, it seems inevitable that T_2 will read the value 1 written by T_1. Yet it is quite possible for T_1 and T_2 to read 0, if the writes remain buffered, or if a compiler chooses to reorder the instructions in both programs.

To avoid complete chaos, every shared memory machine has an underlying *memory (consistency) model*, a contract between a program and the underlying machine architecture that constrains the order in which memory operations become visible to processors [7]. By constraining the order of operations, a memory model determines which values can legally be returned by each read operation. The most common memory model, *sequential consistency* (SC) [22], ensures that memory operations performed by the various processors are *serialized*, or seen in the same order by all processors, and that the common order preserves the order dictated by each program. This results in a model similar to the simple and familiar uniprocessor model. In the example of Fig. 1, it would not be possible for T_1 and T_2 to both read 0 under SC, since the write-before-read order is maintained for both programs.

Because of the requirement that all memory operations be serialized, the SC model is quite restrictive; it requires a high level of interprocessor communication in order to keep caches consistent, and it prohibits most reordering operations at the compiler and hardware levels. Other memory models [12, 9, 3, 28, 8] have been proposed to relax the requirements imposed by SC and therefore allow a wider range of optimizations.

Relaxed memory models place fewer constraints on the memory system than SC, and a smaller set of constraints permits more parallelism and requires less interprocessor communication. However, the results of some program executions may be quite counterintuitive to programmers familiar with uniprocessor or sequential consistency. As program behavior becomes less restricted, reasoning about programs becomes more difficult.

Memory models are subtle, hard to describe, and even harder for the average programmer to understand. The original memory model for the Java programming language [13], written in natural language, suffered from serious imprecision and ambiguity [17]. The possibility of multiple interpretations of the memory model threatens Java's promise of "write once, run anywhere" software. Furthermore, the language designers themselves were victims of their memory model's complexity; they did not realize that their memory model not only disallowed many common optimizations but also failed to guarantee some basic safety properties [25].

There have been two recent attempts to come up with a better model: one by Manson and Pugh [24] and another by Maessen, Arvind and Shen [23] that uses the basic operations of the *Commit-Reconcile-Fence (CRF)* model [28]. We shall call the former the *Manson-Pugh JMM (Java memory model)* and the latter the *CRF JMM*. In both models, the authors realize the importance of precise description. However, the notations describing the two models are quite different. Manson and Pugh donot give a formal semantics for theie notation and use an operational pseudocode reminiscent of ASM, while the CRF JMM is described through a combination of algebraic rules (imposing constraints on instruction reordering) and term-rewriting rules (giving the behavior of the memory system). Furthermore, the levels of abstraction of the two models are

quite different. Manson and Pugh conceive of a history of abstract write operations whose relative ordering determines which values are readable at any given time. Where the values of these abstract writes are stored is intentionally left unspecified. The CRF JMM takes a lower-level view, describing a particular memory system architecture with a cache for each thread and a single common main memory area. Furthermore, Java operations are actually translated into finer-grained CRF operations, and rules are given for legal reorderings of CRF operations. The lower level of abstraction may be more satisfying in its concreteness, but it comes at a cost. For instance, in *cache-only memory architectures* [18, 20], there is no single fixed main memory area; rather, data dynamically move to the threads that access them. In such architectures, the notions of cache and main memory are blurred in a way that is hard to capture in CRF.

In this document, we develop a unified representation of the memory models using the operational semantics methodology of *Abstract State Machines (ASM)* [2][1]. This makes it easier to compare the models. We also define a modified version of the *Location Consistency (LC)* memory model [8], in ASM terms, and prove that it is identical to the Manson-Pugh JMM. Finally, we prove that the CRF JMM is more restrictive than either the Manson-Pugh JMM or the LC JMM.

We provide a brief introduction to ASMs as an appendix. For more information refer to [35] and [36].

[1] Our ASM notation is taken from Gurevich's Lipari Guide [16] and 1997 ASM Guide [15]. We also use the **seq** rule proposed in Börger and Schmid [5].

2 Overview of the new Java memory models

The new Java memory models [23, 24] preserve the fundamental ideas of the original [13]. Each *thread* executes Java bytecode instructions sequentially, with *read (load)* and *write (store)* operations performed on a number of *variables*. A variable is a location at which *values* are stored. In a shared memory system, there may be multiple instantiations of a single variable (*e.g.*, cached by various threads), so there may be multiple values associated with a single variable at any time. The programmer may specify certain variables as *volatile*; such variables obey additional consistency constraints.

Threads may also perform basic synchronization operations. Each object has an associated *monitor*, on which threads perform *lock (enter)* and *unlock (exit)* operations. Lock and unlock instructions are guaranteed to be nested appropriately: a thread may enter a monitor m and then enter another monitor n without exiting m, but in this case it must exit n before exiting m.

2.1 Manson-Pugh JMM

Manson and Pugh describe their JMM in terms of a global system that maintains a history of write operations. The global system operates sequentially: in each step, it atomically executes one instruction from a thread. Execution involves firing an operational-style rule, similar to an ASM rule. Examples are shown in Figure 2.

writeNormal (Write$< v, w, g >$):	overwritten$_t \cup =$ previous$_t(v)$
	previous$_t$ $+= < v, w, g >$
	allWrites $+= < v, w, g >$
readNormal (Variable v):	Choose $< v, w, g >$ from allWrites(v) - overwritten$_t(v)$
	Return w

Figure 2: Sample rules from the Manson-Pugh model [24].

Writes are flagged as *previous* or *overwritten* with respect to a given thread. Manson and Pugh informally describe these terms as follows: "[f]or each thread t, at any given step, overwritten$_t$ is the set of writes that thread t knows are overwritten and previous$_t$ is the set of all writes that thread t knows occurred previously"

[24]. Aside from the anthropomorphic nature of this characterization, it is a little misleading: for instance, it seems unintuitive that t may read a value it did not "know" occurred previously, yet in the Manson-Pugh JMM it is possible for a thread t to read a value written by a write not in previous$_t$.

The significance of previous and overwritten is this: any write marked as overwritten by a thread t does not have a readable value according to t. A local write by t is immediately marked as previous according to t. A remote write is marked as previous according to t only after some synchronization between t and the writer. A write is marked as overwritten according to t only if it has already been marked as previous and another write intervenes (either a local write by t or a remote write followed by synchronization with t). Thus every write considered overwritten by t is also considered previous by t.

Writes are also marked as previous or overwritten with respect to each monitor. A lock operation has the effect of taking all writes that are previous or overwritten with respect to the locked monitor m and marking them similarly with respect to the locking thread t. An unlock has a similar effect, marking all previous or overwritten writes with respect to the unlocking thread t and marking them similarly with respect to the unlocked monitor m.

Volatile variables are different in that there is a single value marked readable for all threads and this globally readable value is the only readable value. The level of consistency is higher; all threads agree on a single value as the unique readable value. Furthermore, volatile variables are similar to threads and monitors in that they mark certain writes as previous or overwritten. Operations on volatile variables are similar to lock and unlock operations. A volatile read, like a lock, takes all writes marked previous or overwritten by the read variable and marks them previous or overwritten by the reading thread. A volatile write, like an unlock, takes all writes marked previous or overwritten by the writing thread and marks them previous or overwritten by the written variable.

Final fields are like constant variables. A variable is final if it instantiates a field designated as final in some object. A value can be assigned to a final field either in the field's declaration or in the object's constructor. The field must be assigned a value by the time the final constructor of the object has terminated. Once a value has been assigned to a final field it cannot be reassigned. Final fields can be accessed without locking, which means they do not have to be reloaded at synchronization points.

2.2 Commit-Reconcile-Fence (CRF) JMM

Java bytecode is translated into a sequence of finer-grained CRF instructions. Each Java instruction corresponds to a particular sequence of CRF instructions. For instance, the CRF translation of a Java *load* instruction includes a *reconcile* instruction (ensuring a fresh value for the load), followed by a *loadl (load-local)* instruction (ensuring that a value is in the thread's cache). A Java store instruction has as its translation a *storel (store-local)* (ensuring that the new value is stored locally) and a following *commit* (ensuring the value is propagated to main memory). *Fence* instructions are also added in the translation to prevent certain kinds of instruction reordering.

Once this translation is done, CRF instruction can be reordered, modulo certain constraints. The abstract machine executes the resulting sequence of pending CRF instructions, each labeled with a unique result tag "r". When an instruction is executed, its result is associated with the result tag in a completion map.

Each thread has of a sequence of pending instructions, a set of completed instructions and a cache which maps addresses to values tagged with a state of either Clean or Dirty. The contents of the cache can be moved atomically to and from the main memory. Cache-to-cache data movement is possible only via the main memory. The model has two kinds of rules: *local rules* that execute CRF instructions and act purely on the cache; and *background rules* that move values to and from the main memory. Figure 3 shows some of these rules. Typically, local rules do little in terms of state updates; their primary task is to block until certain thread-local conditions are met. It is the duty of the background rules to effect the state changes

that allow local operations to complete.

Local Rules:

$(r = Storel(a, v); instr, comp, cache[a := -, -]) \Rightarrow (instr, r = \surd/comp, cache[a := v, Dirty])$

$(r = Loadl(a); instr, comp, cache[a := v, s]) \Rightarrow (instr, r = v/comp, cache[a := v, s])$

$(r = Commit(a, v); instr, comp, cache) \Rightarrow (instr, r = \surd/comp, cache)$
where a is not in $cache$ or a is $Clean$

$(r = Reconcile(a, v); instr, comp, cache) \Rightarrow (instr, r = \surd/comp, cache)$
where a is not in $cache$ or a is $Dirty$

Background Rules:

$(instr, comp, cache[a := v, s]) \Rightarrow (instr, comp, cache)$
where s is $Clean$ (Eject a cache entry)

$(instr, comp, cache)/threads, memory[a := v] \Rightarrow (instr, comp, cache[a := v, Clean])/threads, memory[a := v])$
where $cache$ contains no mapping for a (Fetch a value from main memory)

$(instr, comp, cache[a := v, Dirty])/threads, memory[a := -] \Rightarrow$
$(instr, comp, cache[a := v, Clean])/threads, memory[a := v]$ (Write back a value to main memory)

Figure 3: Sample rules from the CRF JMM [23].

Global states are defined as terms, and updates to global states are represented by rewriting terms, following the style of Shen and Arvind [27]. Examples are shown in Figure 3. This leads to rather unwieldy notation; even though each state change is typically quite local in scope, it requires an entire rewriting of the term. Thus for the nonexpert user, it can be difficult to find where the state is changing at each step. This is in contrast to ASM's locality principle for dynamics [4], which allows us to concentrate only on the changes to the states.

2.3 Location Consistency JMM

The Location Consistency (LC) memory model, proposed by Gao and Sarkar [8], was not proposed specifically as a memory model for Java, but in this paper we define a version of it appropriate for Java. In the model, the state of a memory location is described as a partially ordered set of instruction executions. Whenever an instruction is executed, a record of the operation is added to the partially ordered set of operations on that location.

Chains of operations are formed in two ways: by a thread making a series of local operations, in which case the operation instances are linearly ordered, and by synchronization operations, where the operations of a locking thread are ordered with respect to operations performed by the previous unlocking thread. When a location is read the value returned is the value associated with a maximal write in the partially ordered set for that location. Since the history is a partial order rather than a total order, there may be multiple maximal write values that are legal candidates for a read operation.

LC has been formalized using ASM [31]. However, it is not exactly appropriate to Java, since LC lock operations are performed on locations (variables) rather than monitors, and a lock on a location affects the visibility of writes only on that location [30]. Thus we present a modified version of LC in which synchronization operations affect the visibility of writes to all variables.

3 ASM for the Manson-Pugh Model

In the ASM JMM_{MP}, representing the Manson-Pugh JMM, there are two types of agents. Thread agents (members of the universe Thread) execute Java instructions (members of the universe Instruction). These instructions are of four types: read, write, lock and unlock. A write operation generates a write instance (a member of the universe WriteEvent), which is added to the system history. A variable (a member of the universe Variable) and a value (a member of the universe Value) are associated with the write instance. A read operation generates a read instance (a member of the universe Read), selects a write instance that is legally readable, and reads the value of that write instance. A lock or unlock operation performs a synchronization with a given monitor (a member of the universe Monitor). Monitors are agents that control thread ownership. Figure 4 shows the universe for JMM_{MP}. The functions for this model are described in Figure 5.

Universe	**Description**
WriteEvents	Write Instances
ReadEvents	Read Instances
Threads	Threads
Variables	Variables
Monitors	Monitors
Values	Values that can be assigned to a variable
InstTypes	Java Instruction Types: {Read, Write,Lock,Unlock}
Instructions	Java Instructions

Figure 4: JMM_{MP}: Universes.

As long as a Thread is not waiting for ownership of a monitor, it starts work on its next instruction, given by the monitored function currInst. The functions type, var, monitor, and val give the various parameters associated with each instruction. Read, write, and unlock operations are atomic, while a lock operation may require a complementary action by a Monitor agent, granting ownership to the requested monitor. Each write, lock, and unlock action updates the functions previous? and overwritten? appropriately. The Thread module is given in Fig. 6. In this model we ignore final fields. See section 7 for an ASM description of Final Fields

Function	Profile/Description
currInst	Thread → Instruction Gives the current operation for the thread. (monitored)
type	Instruction → {Read, Write, Lock, Unlock} Returns the type of the given operation.
var	Instruction ∪ WriteEvent → Variable Returns the variable associated with the given operation.
monitor	Instruction → Monitor Returns the monitor associated with the given operation.
val	Instruction ∪ WriteEvent ∪ VolatileVariable → Value Returns the value associated with the given operation/volatile variable.
previous?	WriteEvent × (Thread ∪ Monitor ∪ VolatileVariable) → Boolean Is the given write considered previous by the given thread/monitor?
overwritten?	WriteEvent × (Thread ∪ Monitor ∪ VolatileVariable) → Boolean Is the given write considered overwritten by the given thread/monitor?
locks	Thread × Monitor → Nat Returns the number of locks that a thread has on a monitor.

Figure 5: JMM$_{MP}$: Functions.

Monitors grant exclusive access to one thread at a time, and a thread may build up multiple locks on a single monitor, in which case it needs to perform multiple unlocks before any other thread can be granted a lock on the monitor. If some thread already has a claim on a monitor, then that thread may either increment or decrement its number of locks. If no thread has a claim, the monitor agent may choose a thread to grant a claim to. The Monitor module is given in Fig. 7. When a thread t is granted a lock on a monitor m, then the number of locks by t on monitor m (given by function locks) is incremented.

A write to a volatile variable v updates the globally visible value for v. Similarly, a read of a volatile variable simply uses this unique value; there is no choice involved. The previous? and overwritten? functions are updated similarly to the lock and unlock operations. The special rules for volatile variables are given in Fig. 8.

```
module Thread:
let inst = Self.currInst
    case inst.type of
    Read: Read inst.var
    Write: Write inst.val to inst.var
    Lock: Lock inst.monitor
    Unlock: Unlock inst.monitor


rule Read v:                                              (Perform a read of variable v)
if VolatileVariable(v) then Read volatile v
else
    extend Read with r
        choose w: WriteEvent: w.var = v and not w.overwritten?(Self)
            r.var := v    r.val := w.val


rule Write val to v:                                      (Write value val to variable v)
if VolatileVariable(v) then Write val to volatile v
else
    extend WriteEvent with w
        w.var := v    w.val := val
        w.previous?(Self) := True
    do-forall w: WriteEvent: w.var = v and w.previous?(Self)
        w.overwritten?(Self) := true


rule Lock m:                                              (Get a lock on monitor m)
if Self.locks(m)≠undef then
    Self.locks(m) := Self.locks(m)+1
    do-forall w: WriteEvent: w.previous?(m)
        w.previous?(Self) := true
    do-forall w: WriteEvent: w.overwritten?(m)
        w.overwritten?(Self) := true


rule Unlock m:                                            (Release a lock on monitor m)
Self.locks(m) := Self.locks(m)-1
do-forall w: WriteEvent: w.previous?(Self)
    w.previous?(m) := true
do-forall w: WriteEvent: w.overwritten?(Self)
    w.overwritten?(m) := true
```

Figure 6: JMM$_{MP}$: Thread module.

```
module Monitor:
if (∀t: Thread) t.locks(Self)=undef then
    choose t: Thread
        t.locks(Self) := 0
else
    choose t: Thread: t.locks(Self)=0
        t.locks(Self) := undef
```

Figure 7: JMM$_{MP}$: Monitor module.

```
rule Read volatile v:                          (Perform a read of volatile variable v)
extend Read with r
    r.var := v    r.val := v.val
do-forall w: WriteEvent: w.previous?(v)
    w.previous?(Self) := true
do-forall w: WriteEvent: w.overwritten?(v)
    w.overwritten?(Self) := True
```

```
rule Write val to volatile v:                  (Write value val to volatile variable v)
extend WriteEvent with w
    w.var := v    v.val := v
    do-forall w: WriteEvent: w.previous?(Self)
        w.previous?(v) := true
    do-forall w: WriteEvent: w.overwritten?(Self)
        w.overwritten?(v) := true
```

Figure 8: JMM$_{MP}$: Rules for VolatileVariables.

4 ASM for the CRF Model

Universe	Description
JavaInstructions	Java Instructions: { Load,Store, Enter, Exit }
Values	Values that can be read or written
Threads	Threads
CacheStatus(s)	Cache Entry Status: {Clean,Dirty,Frozen,Invalid}
Variables	Variables
VolatileVariables	Volatile variables: sub-universe of variables
FinalVariables	Final Variables: sub-universe of variables
FenceAddress	Fence Addresses:{$v \cup m \cup$ {*V, *R, *L, *VR, *VRL}}
CRFInstructions	Loadl,Storel,Commit,Reconcile,Fence.Freeze
Monitors(m)	Monitors
InstructionType	Instructions Types: { JavaInstTypes $\cup$ CRFInstTypes }

Figure 9: Universes.

The ASM JMM_{CRF}, representing the CRF JMM, deals with instructions at two levels of abstraction: Java instructions, as in JMM_{MP}, and CRF instructions (members of the universe CRFInstruction). Figure 9 shows the universe for JMM_{CRF}. The functions for this model are described in Figure 10. The CRF JMM uses "Load", "Store", "Enter" and "Exit" as names for the Java instructions, as opposed to "Read", "Write", "Lock" and "Unlock" in the Pugh-Manson JMM. The actions of the ASM can be categorized as follows: add a Java instruction (translating it into appropriate CRF instructions), reorder CRF instructions, execute a CRF instruction, or initiate communication between a cache and main memory through a *background operation*. The Thread module for JMM_{CRF} is given in Fig. 11. In this section we omit final fields. Final fields are discused in section 7.

CRF instructions are of the following types. A *storel (store local)* instruction writes a value to a thread's cache, without affecting main memory. The cache entry is flagged as *dirty*, indicating that its value must eventually be written back to main memory. Functions cacheValue and cacheStatus give the cache state for each thread and variable. The other CRF instructions do not affect the cache directly; rather, they block

Function	Profile/Description
currJavaInst	Thread → JavaInstruction Gives the current instruction for the thread. (monitored)
cacheValue	Thread × Variable → Value Returns the value cached by the given thread for the given variable.
cacheStatus	Thread × Variable → {CacheStatus} Returns the status of the given thread's cache entry for the given variable.
memoryValue	Variable → Value Returns the main memory value of the given variable.
type	JavaInstructions × CRFInstructions → InstructionType Returns the instruction type
var	JavaInstruction ∪ CRFInstruction → Variable Returns the variable associated with the given instruction.
val	JavaInstruction ∪ CRFInstruction → Variable Returns the variable associated with the given instruction.
addInst	Sequence(CRFInstruction) × CRFInstruction → Sequence(CRFInstruction) Adds the given CRF instruction to the end of the given sequence.
swapAt	Sequence(CRFInstruction) × Nat → Sequence(CRFInstruction) Gives the result of swapping the instruction at the given index with its successor.
CRFInsts	Thread → Sequence(CRFInstruction) Returns the sequence of instructions for the given thread.
Loadl, Reconcile, Commit	Variable → CRFInstruction Returns a Loadl/Reconcile/Commit instruction with the given variable parameter.
Lock, Unlock	Monitor → CRFInstruction Returns a Lock/Unlock instruction with the given monitor parameter.
Storel	Variable × Value → CRFInstruction Returns a Storel instruction with the given parameters.
Fence	{R, W} × {R, W} × FenceAddress × FenceAddress → FenceInst Returns a Fence instruction with the given parameters.
addr	CRFInstruction → Variable ∪ Monitor Returns the variable/monitor associated with the given instruction.
preInst, postInst	CRFInstruction → {R,W} Returns the pre/post-instruction type of the fence instruction.
preAddr, postAddr	CRFInstruction → FenceAddress Returns the pre/postaddress of a fence instruction.

Figure 10: JMM$_{CRF}$: Functions.

module Thread:
choose among
Add Java instruction
Reorder CRF instructions
Execute CRF instruction
Perform background operation

Figure 11: JMM$_{CRF}$: Thread module.

```
rule Proceed:                    (Remove current CRF instruction from the list of pending instructions)
Self.CRFInsts := Self.CRFInsts.tail

rule Proceed when cache status not s:    (Proceed when the cache status of the current address is not s)
let v = Self.CRFInsts.head.addr
    if Self.cacheStatus(v) ≠ s then Proceed


rule Execute CRF instruction:
let inst = Self.CRFInsts.head
    case inst.type of
        Storel:     Self.cacheValue(inst.addr) := inst.val
                    Self.cacheStatus(inst.addr) := Dirty
                    Proceed
        Loadl:      Proceed when cache status not Invalid
        Reconcile:  Proceed when cache status not Clean
        Commit:     Proceed when cache status not Dirty
        Fence:      Proceed
        Lock:       if Self.locks(inst.monitor)≠undef then
                        Self.locks(inst.monitor) := Self.locks(inst.monitor)+1
                        Proceed
        Unlock:     Self.locks(inst.monitor) := Self.locks(inst.monitor)-1
                    Proceed
```

Figure 12: JMM$_{CRF}$: Rule for execution of CRF instructions.

until certain cache conditions are met through execution of background operations. A *loadl (load local)* instruction completes only when an entry for the given variable has been loaded into main memory. A *reconcile* instruction completes only when there is no cache entry for the given variable is either nonexistent or dirty. This ensures that subsequent local loads use a relatively up-to-date value. A *commit* instruction completes only when a cache entry for the given variable is either nonexistent or clean. This ensures that any previous local store has had its value written back to main memory. Each CRF instruction has an associated type field. Each storel, loadl, reconcile, commit, lock, and unlock instruction has an addr field, giving the variable or monitor on which the instruction operates. In addition, each Storel instruction has a val field. Finally, a fence instruction has meaning only in terms of restricting certain kinds of instruction reordering; in terms of actual execution, a fence behaves as a no-op. Functions Storel, Loadl, *etc.* give CRF instructions with the appropriate parameters. Figure 12 shows the rule for executing a CRF instruction.

```
rule Perform background operation:
choose v: Variable
    case Self.cacheStatus(v) of
    Invalid:  Self.cacheValue(v) := v.memoryValue
              Self.cacheStatus(v) := Clean
    Clean:    Self.cacheValue(v) := undef
              Self.cacheStatus(v) := Invalid
    Dirty:    Self.cacheStatus(v) := Clean
              v.memoryValue := Self.cacheValue(v)
```

Figure 13: JMM$_{CRF}$: Rule for background operations.

The interaction between cache and main memory that permits completion of these instructions is accomplished through so-called background operations. They do the following: in the case of a dirty cache entry, write the cached value to main memory and mark the entry clean; in the case of a clean cache entry, remove it; in the case of a nonexistent cache entry, create one and give it the current value stored in main memory. The function memoryValue gives the current value in main memory for each variable. Fig. 13 gives the rule for background operations.

rule *Add CRF instructions* $inst_1 \ldots inst_k$:
    Self.CRFInsts := Self.CRFInsts.addInst($inst_1$)
seq ...
seq Self.CRFInsts := Self.CRFInsts.addInst($inst_k$)

rule *Add Java instruction*:
let *inst* = Self.currJavaInst
    case *inst*.type **of**

    Load: **if** VolatileVariable(*inst*.var) **then**
           Add CRF instructions Fence(W,R,*V,*inst*.var), Reconcile(*inst*.var),LoadI(*inst*.var),
                                 Fence(R,R,*inst*.var,*VR), Fence(W,R,*inst*.var,*R)
           else
           Add CRF instructions Reconcile(*inst*.var), LoadI(*inst*.var)

    Store: **if** VolatileVariable(*inst*.var) **then**
           Add CRF instructions Fence(R,W,*VR,*inst*.var),Fence(W,W,*VR,*inst*.var),StoreI(*inst*.var,*inst*.val),
                                 Commit(*inst*.var)
           else
           Add CRF instructions StoreI(*inst*.var,*inst*.val), Commit(*inst*.var)
    Enter:
           Add CRF instructions Fence(W,W,*L,*inst*.monitor), Lock(*inst*.monitor), Fence(W,R,*inst*.monitor,*VR),
                                 Fence(W,W,*inst*.monitor,*VRL)
    Exit:
           Add CRF instructions Fence(W,W,*VR,*inst*.monitor), Fence(R,W,*VR,*inst*.monitor),
                                 Unlock(*inst*.monitor)

Figure 14: JMM$_{CRF}$: Rules for adding CRF instructions.

As a Java instruction is added, it is translated into a sequence of CRF instructions (given by the function CRFInsts). The function addInst returns the CRFInstruction sequence achieved by appending a given CRFInstruction to a given sequence. This sequence can be permuted, with certain exceptions. For instance, a local load instruction on a given variable cannot be executed before a reconcile instruction on the same variable that precedes it in original program order; this would subvert the intent of the reconcile, to ensure that a fresh value is loaded in at the local load. The function swapAt takes a sequence of CRFInstructions and an index i into the sequence and returns the result of swapping the instructions at indices i and $i+1$. Fig. 14 and Fig. 15 give the rules for adding Java instructions and reordering CRF instructions, respectively.

Fence instructions control reordering of other CRF instructions. Each fence has a *pre-instruction type* and a *post-instruction type*. A pre-instruction type of R indicates that the fence prevents certain kinds of read instruction from moving after the fence, while a pre-instruction type of W means that certain writes cannot be moved after the fence. Similarly, the post-instruction type indicates what kinds of instruction (read or write) are prevented from moving before the fence. In addition, a fence has a *preaddress* and a *postaddress*. These restrict the scope of the fence to instructions on certain addresses. A fence (pre/post)address may be

term a.inFenceAddr?(fa): Is address a included in fence address fa?
$a = fa$
or (VolatileVariable(v) and $fa \in$ {*V, *VR, *VRL})
or (Variable(a) and not VolatileVariable(a) and $fa \in$ {*R, *VR, *VRL})
or (Monitor(a) and $fa \in$ {*L, *VRL})

term $inst_1$.fencedBefore?($inst_2$): Is instruction $inst_1$ prevented from moving after (fence) instruction $inst_2$?
$inst_2$.type=Fence and $inst_1$.type $\in$ {Loadl, Commit, Lock, Unlock}
and $inst_1$.addr.inFenceAddr?($inst_2$.preAddr)
and ($inst_1$.type=Loadl $\Rightarrow$ $inst_2$.preInst=R)
and ($inst_1$.type=Commit $\Rightarrow$ $inst_2$.preInst=W)

term $inst_2$.fencedAfter?($inst_1$): Is instruction $inst_2$ prevented from moving before (fence) instruction $inst_2$?
$inst_1$.type=Fence and $inst_2$.type $\in$ {Reconcile, Storel, Lock, Unlock} and $inst_2$.addr.inFenceAddr?($inst_1$.postAddr))
and ($inst_2$.type=Reconcile $\Rightarrow$ $inst_1$.postOp=R)
and ($inst_2$.type=Storel $\Rightarrow$ $inst_1$.postOp=W)

term swappable?($inst_1, inst_2$): Can instruction $inst_1$ be swapped with the following instruction $inst_2$?
not $inst_1$.fencedBefore?($inst_2$) and not $inst_2$.fencedAfter?($inst_1$)
and $inst_1$.type=Loadl $\Rightarrow$ not($inst_2$.type=Storel and $inst_2$.addr=$inst_1$.addr)
and $inst_1$.type=Storel $\Rightarrow$ not($inst_2$.type $\in$ {Loadl, Storel, Commit} and $inst_2$.addr=$inst_1$.addr)
and $inst_1$.type=Lock $\Rightarrow$ not($inst_2$.type=Unlock and $inst_2$.addr=$inst_1$.addr)
and $inst_1$.type=Reconcile $\Rightarrow$ not($inst_2$.type=Loadl and $inst_2$.addr=$inst_1$.addr)

rule *Reorder CRF instructions*:
choose i: Nat: Self.CRFInsts.at(i+1)$\neq$undef and swappable?(Self.CRFInsts.at(i), Self.CRFInsts.at(i+1))
    Self.CRFInsts := Self.CRFInsts.swapAt(i)

Figure 15: JMM$_{CRF}$: Rules for reordering CRF instructions.

a single variable or monitor, or it may be a wildcard reference such as *V, *VR, or *VRL. The fence address *V refers to "all volatile variables", *VR to "all (regular and volatile) variables", and *VRL to "all variables and locks". Thus for instance, a fence with pre-instruction type W, post-instruction type R, preaddress *V, and postaddress v (for some variable v) disallows movement of preceding write instructions on volatile variables after the fence and disallows movement of following read instructions on variable v before the fence. The FenceAddress universe is defined as Variable $\cup$ Monitor $\cup$ {*V,*R,*L,*VR,*VRL}. Each Fence instruction has associated preInst, postInst, preAddr, and postAddr fields.

5 ASM for the Location Consistency JMM

Universe	Description
WriteEvents	Write instances in system history
ReadEvents	Read instances
Threads	Threads
Variables	Variables
Monitors	Monitors of Threads
Values	Values that can be assigned to a variable
Locks	Locks
Unlocks	Unlocks

Figure 16: JMM_{LC}: Universes.

Function	Profile/Description
$\prec$	Event $\times$ Event $\rightarrow$ Boolean Precedence relation between events.
latest	Thread $\times$ Variable $\rightarrow$ Event Returns the latest event issued by the given thread on the given variable.
latestUnlock	Monitor $\rightarrow$ UnlockEvent Returns the latest unlock event issued by the given monitor.

Figure 17: JMM_{LC}: Functions.

We restrict the Location Consistency memory to make it appropriate to Java. The resulting ASM is called JMM_{LC}. The ASM is a modification of JMM_{MP}, involving modifications to the rules for reads, writes, locks and unlocks. The JMM_{LC} uses the functions from JMM_{MP}. Additional functions that we define for

rule *Order* e *after* d: (Make event e a successor of event d, in the relation $\prec$)
if $d \neq$ undef **then** $d \prec e :=$ true

rule *Read* v:
extend ReadEvent **with** rd
  choose w: Write: w.var $= v$ and not $(\exists w'$: WriteEvent$)$ $w \prec^+ w' \prec^*$ Self.latest(w.var)
    rd.val $:= w$.val

rule *Write val to* v:
extend WriteEvent **with** w
  w.var $:= v$ w.val $:= val$
  Order w *after* Self.latest(v)
  Self.latest(v) $:= w$

rule *Lock* m:
if Self.locks(m)$\neq$undef **then**
  Self.locks(m) $:=$ Self.locks(m)+1
  extend LockEvent **with** ℓ
    Order ℓ *after* m.latestUnlock
    do-forall v: Variable
      Order ℓ *after* Self.latest(v)
      Self.latest(v) $:= \ell$

rule *Unlock* m:
extend UnlockEvent **with** u
  Self.locks(m) $:=$ Self.locks(m)-1
  do-forall v: Variable
    Order u *after* t.latest(v)
    Self.latest(v) $:= \ell$
  m.latestUnlock $:= u$

Figure 18: JMM$_{LC}$: Rules for thread operations.

JMM_{LC} are shown in Fig. 17. The modified rules are shown in Fig. 18. Like the LC model, in JMM_{LC} the memory system is represented as an order $\prec$ on operation instances or *events* (members of the universe Event). As operations complete, events are generated and added to the order. The events issued by a single thread are ordered linearly. With a write to a given variable, a new write event is issued and ordered after the writing thread's local events on that variable. The function latest maintains the most recently issued events by each thread on each variable.

Events ordered by different threads may be ordered through lock and unlock actions. Each lock or unlock operation is seen as acting on all variables, so a lock or unlock event is ordered with respect to all events issued by the thread. This is where the LC JMM differs from the general LC model: an acquire or release operation in LC is performed on a particular location, and the resulting event is ordered only with regard to local events on that location. Similarly to the general model, each lock event on a given monitor is ordered after the previous unlock event on that monitor. The function latestUnlock maintains the most recently issued unlock events for each monitor.

Like the Manson-Pugh JMM, reading involves a choice of a legal write event. Here the order $\prec$ determines the legality of each write event. We use $\prec^+$ and $\prec^*$ to represent the transitive closure of $\prec$ and the reflexive, transitive closure of $\prec$, respectively. As long as there is no chain of the form $w \prec^+ w' \prec^* e_t$, where w and w' are write events and e_t is an event performed by thread t, w is a readable value according to t. Read events have no effect on the readability of subsequent write events, so they are not added to the order.

6 Comparisons of the Java memory models

We now use our ASMs to reason about the behavior of the JMMs. We restrict our attention to non-volatile variables. We first show that the Manson-Pugh and LC JMMs are identical in that they allow exactly the same possible behaviour. Next, we show that the CRF JMM is more restrictive than either the Manson-Pugh or the LC JMM.

6.1 Manson-Pugh *vs.* LC

In both the Manson-Pugh and LC JMMs, the synchronization operations (lock and unlock) ensure that other threads are informed of previous write operations. However, threads may share values without synchronization. In a system with a centralized main memory, this may occur through the normal actions of cache writeback and fetch. The fact that there is no synchronization forcing these actions from happening does not imply that they do not happen. Note that in the absence of all synchronization, *all* writes are visible to a thread (except any writes of its own that it has subsequently overwritten). Indeed,it is more accurate to think of synchronization operations *restricting* the visibility of certain writes by rather than making Writes visible. When writes are interleaved with synchronization operations, the resulting chains of write and synchronization operations may cause further writes to be considered overwritten. We introduce the notion of *synchronization chain* to capture this idea.

A synchronization chain begins with a write w and ends with an action by a thread or monitor x. (We express this as $w \rightsquigarrow x$.) The thread or monitor x is informed of the existence of w through a subsequent chain of synchronization operations, including an unlock action by the writer of w and a subsequent unlock operation on x (if x is a monitor) or lock operation by x (if x is a thread). (A trivial synchronization chain exists between w and the thread that performed w.) If another write w' forms part of the chain containing w (which we express as $w \rightsquigarrow w'$), w is considered overwritten by w' and is not a readable value.

Definition. Let w be a write event issued at a move W by a thread s. Let Y be any move after W.

- For thread t, $w \rightsquigarrow t$ at Y if
  - $s = t$, or
  - for some Monitor m, $w \rightsquigarrow m$ at a move X in (W,Y), and t locks m at X.

- For monitor m, $w \rightsquigarrow m$ at Y if
  - s unlocks m at a move X in (W,Y), or
  - for some Thread t, $w \rightsquigarrow t$ at a move X in (W,Y), and t unlocks m at X.
- For any WriteEvent w' issued by a Thread t at a move X, $w \rightsquigarrow w'$ if $w \rightsquigarrow t$ at X.

Lemma 1 *In a run of JMM_{MP}, let w be a write event issued at a move W, and let Y be any move after W. For any thread or monitor x, w.previous?(x) at Y if and only if $w \rightsquigarrow x$ at Y.*

Proof. By induction on the number of moves between W and Y.

Base step: The interval (W, Y) is empty. At W, w.previous?(s) is updated to true, and w.previous?(x) is not updated to true for any other x. Hence, w.previous?(x) at Y only if $x = s$. Furthermore, since there is no lock or unlock move in (W, Y), $w \rightsquigarrow x$ only if $x = s$.

Inductive step: The interval (W, Y) is nonempty. Let X be the last move before Y, and assume that the claim holds at X.

Assume to the contrary that at Y, w.previous?(x) but $w \not\rightsquigarrow x$. Then $w \not\rightsquigarrow x$ at X, and by the inductive hypothesis,not w.previous?(x). Therefore w.previous?(x) must be updated to true at X. By examination of the ASM rules, there are two ways in which this can happen:

- A lock by (Thread) x on a Monitor m such that w.previous?(m) at X. By the inductive hypothesis, $w \rightsquigarrow m$. Then $w \rightsquigarrow x$ at Y.
- An unlock by a Thread t on (Monitor) x such that w.previous?(t) at X. By the inductive hypothesis, $w \rightsquigarrow t$. Then $w \rightsquigarrow x$ at Y.

In either case, $w \rightsquigarrow x$ at Y, contradicting our assumption.

Conversely, assume that at Y, $w \rightsquigarrow x$ but not w.previous?(x). Then not w.previous?(x) at X, and by the inductive hypothesis, $w \not\rightsquigarrow x$. Therefore a synchronization chain from w to x must be created at X. There are two ways in which this can happen:

- A lock by (Thread) x on a Monitor m such that $w \rightsquigarrow m$. By the inductive hypothesis, w.previous?(m) at X, so w.previous?(x) := true at X.
- An unlock by a Thread t on (Monitor) x such that $w \rightsquigarrow t$. By the inductive hypothesis, w.previous?(t) at X, so w.previous?(x):= true at X.

In either case, w.previous?(x) at Y, contradicting our assumption. □

Lemma 2 *In a run of JMM_{MP}, let w be a* WriteEvent *issued at a move W, and let Y be any move after W. For any* Thread *or* Monitor *x, w.overwritten?(x) at Y if and only if there is a* WriteEvent *w' such that $w \rightsquigarrow w'$ and $w' \rightsquigarrow x$ at Y.*

Proof. By induction on the number of moves between W and Y.

Base step: The interval (W, Y) is empty. At W, w.overwritten?(y) is not updated to true for any y, so not w.overwritten?(x) at Y. Furthermore, there is no write w' issued in (X, Y).

Inductive step: The interval (W, Y) is nonempty. Let X be the last move before Y, and assume that the claim holds at X.

Assume that at Y, w.overwritten?(w) but there is no WriteEvent w' as described. Then there is also no such w' at X, so by the inductive hypothesis, not w.overwritten?(x) at X. Thus w.overwritten?(w) := true at X. There are three ways in which w.overwritten?(x) can be updated to true:

- a write by (Thread) x, issuing a WriteEvent w'. In this case, w.overwritten?(x) := true only if w.previous?(x). By Lemma 1, $w \rightsquigarrow x$ at X. Furthermore, by definition $w' \rightsquigarrow x$ at Y.
- a lock by (Thread) x on a Monitor m such that w.overwritten?(m) at X. By the inductive hypothesis, there is a WriteEvent w' issued by a Thread t at W' in (W, X) such that $w \rightsquigarrow t$ at W' and $w' \rightsquigarrow m$ at X. Then at Y, $w' \rightsquigarrow x$.
- an unlock by a Thread t on (Monitor) x such that w.overwritten?(t) at X. By the inductive hypothesis, there is a WriteEvent w' issued by a Thread s at W' in (W, X) such that $w \rightsquigarrow s$ at W' and $w' \rightsquigarrow t$ at X. Then at Y, $w' \rightsquigarrow x$.

In each case, there is a WriteEvent w' for which $w \rightsquigarrow w' \rightsquigarrow x$, contradicting our assumption.
Conversely, assume that at Y, there is a WriteEvent w' such that $w \rightsquigarrow w' \rightsquigarrow x$, but not w.overwritten?(x).

Then there is no such w' at X, so the move at X must establish $w \rightsquigarrow w' \rightsquigarrow x$ for some w'. There are three ways in which this can happen:

- a write by (Thread) x (issuing WriteEvent w') such that $w \rightsquigarrow x$ at X. (In this case, $w' \rightsquigarrow x$ trivially at Y.) By Lemma 1, w.previous?(x) at X, so w.overwritten?(x) := true at X.
- a lock by (Thread) x on a Monitor m, where there is a WriteEvent w' with $w \rightsquigarrow w' \rightsquigarrow m$. (In this case, $w' \rightsquigarrow x$ at Y.) By the inductive hypothesis, w.overwritten?(m) at X, so w.overwritten?(w) := true at X.
- an unlock by a Thread t on (Monitor) x, where there is a WriteEvent w' with $w \rightsquigarrow w' \rightsquigarrow t$. (In this case, $w' \rightsquigarrow x$ at Y.) By the inductive hypothesis, w.overwritten?(t) at X, so w.overwritten?(x) := true at X.

In each case, w.overwritten?(x) at Y, contradicting our assumption. □

Next, we show that the notions of "previous" and "overwritten" have counterparts in the LC JMM. A write is considered previous by a thread or monitor if it precedes the latest event by the thread/monitor, according to $\prec$. A write is considered overwritten if there is a chain including the write and a local event by the thread/monitor that includes another, intervening write.

term w.LC-previous?(x):
Thread(x) $\Rightarrow$ $w \prec^* x$.latest(w.var)
and Monitor(x) $\Rightarrow$ $w \prec^+ x$.latestUnlock

term w.LC-overwritten?(x):
Thread(x) $\Rightarrow$ ($\exists w'$: WriteEvent) $w \prec^+ w' \prec^* x$.latest(w.var)
and Monitor(x) $\Rightarrow$ ($\exists w'$: WriteEvent) $w \prec^+ w' \prec^+ x$.latestUnlock

Lemma 3 *In a run of JMM_{LC}, let w be a* WriteEvent *issued at a move W, and let Y be any move after W. For any* Thread *or* Monitor *x, w.LC-previous?(x) at Y if and only if $w \rightsquigarrow x$ at Y.*

Proof. By induction on the number of moves between W and Y. Let s be the Thread that issues w.

Base step: The interval (W, Y) is empty. For any Thread or Monitor x, w.LC-previous?(x) at Y only if x is a Thread and $w =$ latestLocalEvent(x, w.var), in which case $s = x$ and hence $w \rightsquigarrow x$ at Y. Conversely, since there are no synchronization actions in (W, Y), $w \rightsquigarrow x$ at Y only if $s = x$, in which case $w =$ latestLocalEvent(x, w.var) and hence w.LC-previous?(x).

Inductive step: The interval (W, Y) is nonempty. Let X be the last move before Y, and assume that the claim holds at X.
Assume to the contrary that at Y, w.previous?(x) but $w \not\rightsquigarrow x$ at Y. Then $w \not\rightsquigarrow x$ at X, so by the inductive hypothesis, not w.previous?(w) at X. Thus w.previous?(w) must become true at X. There are only two ways in which this can occur:

- a lock by (Thread) x on a Monitor m, such that $w \prec m$.latestUnlock. By the inductive hypothesis, $w \rightsquigarrow m$ at X. Then at Y, $w \rightsquigarrow x$ at X.
- an unlock by a Thread t on (Monitor) x, such that $w \preceq$ latestLocalEvent(t, w.var). By the inductive hypothesis, $w \rightsquigarrow t$ at X. Then at Y, $w \rightsquigarrow x$ at X.

In either case, $w \rightsquigarrow x$ at Y, contradicting our assumption.
Conversely, assume to the contrary that at Y, $w \rightsquigarrow x$ but not w.previous?(w). Then not w.previous?(w) at X, so by the inductive hypothesis, $w \not\rightsquigarrow x$. Thus a synchronization chain from w to x must be created at X. There are two ways in which this can happen:

- A lock by (Thread) x on a Monitor m such that $w \rightsquigarrow m$. Let ℓ be the LockEvent issued at X. By the inductive hypothesis, w.previous?(w) at X, so $w \prec \ell :=$ true and latestLocalEvent(x, w.var) $:= \ell$ at X.
- An unlock by a Thread t on (Monitor) x such that $w \rightsquigarrow t$. Let u be the UnlockEvent issued at X. By the inductive hypothesis, w.previous?(t) at X, so $w \prec u :=$ true and x.latestUnlock $:= u$ at X.

In either case, w.previous?(w) at Y, contradicting our assumption. □

Lemma 4 *In a run of JMM_{LC}, let w be a* WriteEvent *issued at a move W, and let Y be any move after W.*

- *For any* Thread t, w.LC-overwritten?(t) *at Y if and only if there is a* WriteEvent w' *such that $w \rightsquigarrow w'$ and $w' \rightsquigarrow t$ at Y.*
- *For any* Monitor m, w.LC-overwritten?(m) *at Y if and only if there is a* WriteEvent w' *such that $w \rightsquigarrow w'$ and $w' \rightsquigarrow m$ at Y.*

Proof. By induction on the number of moves between W and Y.

Base step: The interval (W, Y) is empty. There is no WriteEvent issued in (X, Y), and there is no w' for which $w \prec w' :=$ true at X, so there is no Thread or Monitor x for which w.LC-overwritten?(x) at Y.

Inductive step: The interval (W, Y) is nonempty. Let X be the last move before Y, and assume that the claim holds at X.

Assume to the contrary that at Y, w.LC-overwritten?(x) for some Thread or Monitor x, but there is no WriteEvent w' as described. Then there is also no such w' at X, so by the inductive hypothesis, not w.LC-overwritten?(x) at X. There are three ways in which w.LC-overwritten?(w) can come to be true at Y:

- a write by (Thread) x, issuing a WriteEvent w', where w.LC-previous?(x). (In this case, $w \prec w' :=$ true and latestLocalEvent(x, w.var) $:= w'$ at X, so we have $w \prec w' =$ latestLocalEvent(x, w.var) at Y.) By Lemma 3, $w \rightsquigarrow x$ at X. Furthermore, by definition $w' \rightsquigarrow x$ at Y.

- a lock by (Thread) x on a Monitor m such that w.LC-overwritten?(m) at X. (In this case, $w \prec w' \prec m$.latestUnlock at X for some WriteEvent w'. At X, a LockEvent ℓ is issued, $w' \prec \ell :=$ true, and latestLocalEvent(x, w.var) $:= \ell$, so we have $w \prec w' \prec$ latestLocalEvent(x, w.var).) By the inductive hypothesis, there is a WriteEvent w' issued by a Thread t at W' in (W, X) such that $w \rightsquigarrow t$ at W' and $w' \rightsquigarrow m$ at X. Then at Y, $w' \rightsquigarrow x$.
- an unlock by a Thread t on (Monitor) x such that w.LC-overwritten?(w) at X. (In this case, $w \prec w' \prec$ latestLocalEvent(t, w.var) at X for some WriteEvent w'. At X, an UnlockEvent u is issued, $w' \prec u :=$ true, and x.latestUnlock $:= u$, so we have $w \prec w' \prec m$.latestUnlock.) By the inductive hypothesis, there is a WriteEvent w' issued by a Thread s at W' in (W, X) such that $w \rightsquigarrow s$ at W' and $w' \rightsquigarrow t$ at X. Then at Y, $w' \rightsquigarrow x$.

In each case, there is a WriteEvent w' for which $w \rightsquigarrow w' \rightsquigarrow x$, contradicting our assumption.

Conversely, assume to the contrary that at Y, there is a WriteEvent w' as described but not w.LC-overwritten?(w).

Then not w.LC-overwritten?(x) at X, so by the inductive hypothesis, there is no WriteEvent w' such that $w \rightsquigarrow w' \rightsquigarrow x$ at X. Hence the move at X must establish $w \rightsquigarrow w' \rightsquigarrow x$ for some w'. There are three ways in which this can happen:

- a write by (Thread) x (issuing WriteEvent w') such that $w \rightsquigarrow x$ at X. (In this case, $w' \rightsquigarrow x$ trivially at Y.) By Lemma 3, w.LC-previous?(x) at X, so $w \prec w' :=$ true and latestLocalEvent(x, w.var) $:= w'$ at X.
- a lock by (Thread) x on a Monitor m, where there is a WriteEvent w' with $w \rightsquigarrow w' \rightsquigarrow m$. (In this case, $w' \rightsquigarrow x$ at Y.) By the inductive hypothesis, w.overwritten?(m) at X, so $w \prec \ell :=$ true and latestLocalEvent(x, w.var) $:= \ell$ at X.
- an unlock by a Thread t on (Monitor) x, where there is a WriteEvent w' with $w \rightsquigarrow w' \rightsquigarrow t$. (In this case, $w' \rightsquigarrow x$ at Y.) By the inductive hypothesis, w.overwritten?(t) at X, so $w \prec u :=$ true and x.latestUnlock $:= u$ at X.

In each case, w.LC-overwritten?(x) at Y, contradicting our assumption. □

Theorem 1 *If a value is not readable in a run of JMM_{LC}, then that value is not readable in an equivalent run of JMM_{MP}.*

First consider a run of JMM_{LC}, in which Thread t performs a read Rd of variable v, but there is a write event w on v that is not readable.

According to the read rules for the LC model, w is not readable only if w.LC-overwritten?(t) at Rd.
By Lemma 4, w.overwritten?(t) only if there is a write w' such that $w \rightsquigarrow w' \rightsquigarrow t$ at Rd.
By Lemma 2, w.overwritten?(t)at Rd at any equivalent run in JMM_{MP}.
Hence at Rd, w is not read in any equivalent run of JMM_{MP}.

Theorem 2 *If a value is not readable in a run of JMM_{MP}, then that value is not readable in an equivalent run of JMM_{LC}.*

First consider a run of JMM_{MP}, in which Thread t performs a read Rd on variable v but there is some write event w on v which is not readable.

According to the read rules for the JMM_{MP} model, w is not readable only if w.overwritten?(t) at Rd.
By Lemma 2, w.overwritten?(t) only if there is a write w' such that $w \rightsquigarrow w' \rightsquigarrow t$ at Rd.
By Lemma 4, w.LC-overwritten?(t) at Rd.

Hence according to the read rules of the JMM_{LC} model, at Rd, w is not read in any equivalent run of JMM_{LC}.

Finally, we note that any write is readable in JMM$_{LC}$ as long as LC-overwritten? evaluates to false, and any write is readable in JMM$_{MP}$ as long as overwritten? evaluates to false. Let runs of JMM$_{LC}$ and JMM$_{MP}$ be equivalent if they involve the same operations by the same threads/monitors in the same order. We conclude the following:

Corollary *A value is readable in JMM$_{LC}$ if and only if that value is readable in an equivalent run of JMM$_{MP}$.*

6.2 Manson-Pugh/LC *vs.* CRF

Having established the equivalence of the Manson-Pugh JMM and the LC JMM, we now compare these two models to the CRF JMM. Here we face an additional level of complexity. In all three models, each thread services a sequence of Java instructions. But in the CRF JMM, the actions of reading and writing are done at the level of CRF instructions, which are derived from Java instructions and then possibly reordered. In reasoning about the values actually written and read (to local caches) in the CRF JMM, we must look at the level of CRF instructions. But to provide a comparison to the other models, we must relate the relevant CRF instructions back to the Java instructions that triggered them, bearing in mind that instruction reordering may occur between the issuing of the Java instruction and the execution of the CRF instruction.
To this end, we introduce some notation. Let m be a move at which a CRF instruction is executed. This instruction was generated at an earlier move, in which a Java instruction was issued and translated into CRF. We refer to the earlier move as $\overline{m}$. We then establish that the CRF JMM is at least as constrained as the other two models.

Lemma 5 *Let ll be a local load in a run of JMM$_{CRF}$, and let sl be the local store that wrote the value read at ll. Then there is no store operation S such that $\overline{sl} \rightsquigarrow S \rightsquigarrow t$ at $\overline{ll}$.*

Let ll be a loadl of Variable v by Thread t. We trace the value read at ll to a previous storel operation sl and then show that there can be no store operation S such that $\overline{sl} \rightsquigarrow S \rightsquigarrow t$ at $\overline{ll}$. There are three cases:

Case 1. At ll, cacheStatus(t, v)=dirty.

Case 2. At ll, cacheStatus(t, v)=clean.

- **Case 2a.** The last move before ll at which cacheStatus(t, v) := clean is a (t, v) writeback wb.
- **Case 2b.** The last move before ll at which cacheStatus(t, v) := clean is a fetch f on v by t.

In case 1, there must be a (t, v) storel before ll at which cacheStatus(t, v) := dirty; let sl be the last one. In (sl, ll), there is no (t, v) writeback, since this would update cacheStatus(t, v) := clean. Only such a writeback can update cacheStatus(t, v) := clean, and $commit(\overline{sl})$ occurs only if cacheStatus(t, v)=clean. Therefore, $commit(\overline{sl})$ does not occur in (sl, ll).

In case 2a, there is no (t, v) fetch updating cacheStatus(t, v) := clean in (wb, ll). wb occurs only if cacheStatus(t, v)=dirty; let sl be the last (t, v) storel before wb to update cacheStatus(t, v) := dirty. Since cacheStatus(t, v)=clean at $commit(\overline{sl})$, wb must precede $commit(\overline{sl})$.

In cases 1 and 2a, sl and ll are performed by t, and there is no storel by t on v in (sl, ll). Since $sl \ll ll$, it must be that $\overline{sl} < \overline{ll}$, and no store S to v by t can occur in $(\overline{sl}, \overline{ll})$, since $storel(S)$ is either $\ll sl$ or $\gg ll$. Hence there is no store S by t such that $\overline{sl} \rightsquigarrow S$, so the only way in which $\overline{sl} \rightsquigarrow S \rightsquigarrow t$ for any store S is if S is performed by a Thread other than t. For this to happen, there must be an unlock U and a lock L by t such that $S < U < L < \overline{ll}$. In this case we have the following contraints,

- $sl \ll commit(\overline{sl}) \ll prefence_{ww}(U) \ll unlock(U)$, and
- $lock(L) \ll postfence_{wr}(L) \ll reconcile(\overline{ll}) \ll ll$,

so $sl < commit(\overline{sl}) < reconcile(\overline{ll}) < ll$.

In case 1 these constraints cannot be met, since $commit(\overline{sl})$ does not occur in (sl, ll).

In case 2a, at $reconcile(\overline{ll})$, cacheStatus(t, v) must be either dirty or undef. Because cacheStatus$(t, v) :=$ clean at wb and there is no storel in $(wb, reconcile(\overline{ll}))$ to update cacheStatus$(t, v) :=$ dirty, it must be that cacheStatus(t, v)=undef at $reconcile(\overline{ll})$. At ll, cacheStatus$(t, v) \neq$ undef, but since there is no storel or fetch in $(reconcile(\overline{ll}), ll)$ to update cacheStatus$(t, v)$to a non-undef value the above constraints cannot be met. Hence in cases 1 and 2a there is no store operation S such that $\overline{sl} \rightsquigarrow S \rightsquigarrow t$.

In case 2b, there is no (t, v) writeback updating cacheStatus$(t, v) :=$ clean in (f, ll). At f, CV$(t, v) :=$ v.memVal. Let wb be the last writeback of v (by some thread r) updating v.memVal before f. At wb, cacheStatus(r, v)=dirty; let sl be the latest storel (at which cacheStatus$(r, v) :=$ dirty) before wb.
Assume there is a store S such that $\overline{sl} \rightsquigarrow S \rightsquigarrow t$. Let s be the Thread that performs S. We show that $storel(S)$ does not occur in (sl, ll), and therefore it cannot be the case that $\overline{sl} \rightsquigarrow S \rightsquigarrow t$. There are three cases:

- $r = s = t$. Then since sl is the last (t, v) storel before wb, $storel(S)$ is not in (sl, wb).

  Furthermore, $storel(S)$ is not in (wb, f). If $storel(S)$ did occur in this interval, then at $storel(S)$, cacheStatus$(t, v) :=$ dirty, but at f, cacheStatus(t, v)=undef. Hence there would have to be a (t, v) eject j in $(storel(S), f)$. At j, cacheStatus(t, v)=clean, which would require a (t, v) writeback in $(storel(S), j)$. However, since wb is the last writeback to v before ll, this is impossible.

  Finally, $storel(S)$ is not in (f, ll). If $storel(S)$ did occur in this interval, then at $storel(S)$, cacheStatus(t, v) := dirty, but cacheStatus(t, v)=clean at ll. Hence there would have to be a (t, v) writeback in $(storel(S), ll)$ (at which cacheStatus$(t, v) :=$ clean). This is impossible since there is no (t, v) writeback in (f, ll).

- $r = s \neq t$. Since sl is the last (t, v) storel before wb, $storel(S)$ is not in (sl, wb). Furthermore, $storel(S)$ is not in (wb, ll). If it did occur in this interval, there must be an unlock U by r followed by a lock L by t such that $S < U < L < \overline{ll}$. In this case,

  - $storel(S) \ll commit(S) \ll prefence_{ww}(U) \ll unlock(U)$, and
  - $lock(L) \ll postfence_{wr}(L) \ll reconcile(\overline{ll}) \ll ll$,

  so $wb < storel(S) < commit(S) < ll$. At $storel(S)$, cacheStatus$(r, v) :=$ dirty, but at $commit(S)$, cacheStatus(t, v)=clean, so there would have to be an (r, v) writeback in $(storel(S), commit(S))$ to update cacheStatus$(r, v) :=$ clean. This is impossible since wb is the last writeback to v before ll.

- $r \neq s \neq t$. Then there must be an unlock U_r by r, a lock L_s and unlock U_s by s, and a lock L_t by t such that $\overline{sl} < U_r < L_s < S < U_s < L_t < \overline{ll}$. We have

  - $sl \ll commit(\overline{sl}) \ll prefence_{ww}(U_r) \ll unlock(U_r)$
  - $lock(L_s) \ll postfence_{ww}(L_s) \ll storel(S) \ll commit(S) \ll prefence_{ww}(U_r) \ll unlock(U_r)$, and
  - $lock(L_t) \ll postfence_{wr}(L_t) \ll reconcile(\overline{ll}) \ll ll$,

  so $sl < commit(\overline{sl}) < storel(S) < commit(S) < reconcile(\overline{ll}) < ll$.

  First, $storel(S)$ is not in (wb, ll). If it did occur in this interval, then cacheStatus$(t, v) :=$ dirty at $storel(S)$ but cacheStatus(t, v)=clean at ll, so there would have to be a (t, v) writeback in $(storel(S), ll)$, to update cacheStatus$(t, v) :=$ clean. This is impossible since wb is the last writeback to v before ll.

Furthermore, $storel(S)$ is not in (sl, wb). If it did occur in this interval, $unlock(U_r)$ would have to be in $(sl, storel(S))$ and $commit(\overline{sl})$ would have to be in $(sl, unlock(U_r))$. At $storel(S)$, cacheStatus(t, v) := dirty, but at $commit(\overline{sl})$, cacheStatus(t, v)=clean, so there must be a (t, v) writeback in $(sl, commit(\overline{sl}))$ to update cacheStatus(t, v) := clean. This is impossible since wb is the last writeback to v before ll.

In Case 1 and Case 2, since there is no S for which $\overline{sl} \rightsquigarrow S \rightsquigarrow t$ at ll, $\overline{sl}$ is readable at $\overline{ll}$ according to JMM_{LC} and JMM_{PM}. □

Corollary *A value is readable in JMM_{CRF} only if that value is readable in an equivalent run of JMM_{MP}.*

The converse of the above corollary is not true, as can be shown by a simple example. Let T_1 and T_2 be Threads accessing a common Variable x. The Threads execute the following instruction sequences:

T_1**:** write$(x, 1)$; read(x); read(x)

T_2**:** write$(x, 2)$

Assume that x=0 initially. It is easy to show that in any run of JMM_{CRF} with the above sequences of Java instructions, if T_1 first reads 2, it cannot then read 1. First, note that T_1's Storel and Loadl instructions follow the program order of the original Java instructions, since the swappable? predicate prevents the Storel from being ordered after the initial Loadl.

At T_1's first loadl, T_1.cacheValue(x)=2. This is only possible if a fetch (background) operation earlier updated T_1.cacheValue(x) := x.memoryValue. At the time of this fetch, T_1.cacheStatus(x)=Invalid. Thus an earlier writeback (background) operation must have updated T_1.cacheStatus(x) := Clean, followed by an ejection updating T_1.cacheStatus(x) := Invalid. After this writeback, T_1.cacheStatus(x)=Clean, so T_1 never performs another writeback. Since x.memoryValue=2 at the time of T_1's fetch, a writeback operation must have updated x.memoryValue := T_2.cacheValue(x), and this writeback must have followed T_1's writeback. Thus x.memoryValue remains 2, and the value 1 is no longer available to read.

No such constraint exists for the Manson-Pugh JMM. Let w_1 and w_2 be the WriteEvents issued by T_1 and T_2, respectively. Since there is no synchronization between T_1 and T_2, $w_2 \not\rightsquigarrow T_1$ at the time of T_1's second read, so there is no w' such that $w_1 \rightsquigarrow w' \rightsquigarrow T_1$. By Lemma 2, w_1 is readable.

7 Semantics for Final Fields.

Figure 19shows the additional functions for the final field semantics in JMM_{MP}. knownfrozen represents the set of writes to final fields that are known to be frozen. The function FinalValue returns the value associated with the. The function defaultValue returns the defaultValue associated with a final field. Figure 20 and Figure 21 show the execute rules and the rules for the final fields in JMM_{CRF}.

Function	**Profile/Description**
knownFrozen	Threads ∪ VolVar ∪ Monitor → Boolean Returns true if the variable is known to be frozen
FinalValue	Variable → Value Returns the final value for given variable
defaultValue	Variable → Value Returns the default value for the given variable

Figure 19: JMM_{MP}: Functions for final fields.

rule *WriteFinal val to v*: (Write final value *val* to variable v)
    extend WriteEvent **with** w
        w.var := v
        v.finalValue := val

rule *FreezeFinal v*: (Freeze the final value to variable v)
    do-forall w: WriteEvent: w.overwritten?(self)
        w.overwritten?(v) := true
    v.knownFrozen($self$) := True

rule *ReadFinal* v: (Perform a read of final variable v)
    extend ReadEvent **with** r
    r.var := v
    if v.knowFrozen? **then**
    do-forall w: WriteEvent: w.overwritten?(v)
        w.overwritten?(self) := true
        r.val := v.finalValue
    else
        ChooseAmong
            r.val := v.finalValue
            r.val := v.defaultValue
    endif

rule *Lock* m: (Get a lock on monitor m)
if Self.locks(m)≠undef **then**
    Self.locks(m) := Self.locks(m)+1
    do-forall w: WriteEvent: w.previous?(m)
        w.previous?(Self) := true
    do-forall w: WriteEvent: w.overwritten?(m)
        w.overwritten?(Self) := true
    do-forall w: WriteEvent: w.knownFrozen?(m)
        w.knowFronzen?(Self) := true

rule *Unlock* m: (Release a lock on monitor m)
Self.locks(m) := Self.locks(m)-1
do-forall w: WriteEvent: w.previous?(Self)
    w.previous?(m) := true
do-forall w: WriteEvent: w.overwritten?(Self)
    w.overwritten?(m) := true
do-forall w: WriteEvent: w.knownFrozen?(Self)
    w.knownFrozen?(m) := true

rule *Read volatile* v: (Perform a read of volatile variable v)
extend Read **with** r
    r.var := v r.val := v.val
do-forall w: WriteEvent: w.previous?(v)
    w.previous?(Self) := true
do-forall w: WriteEvent: w.overwritten?(v)
    w.overwritten?(Self) := True
do-forall w: WriteEvent: w.knownFrozen?(v)
    w.knownFrozen?(Self) := true

rule *Write val to volatile* v: (Write value *val* to volatile variable v)
extend WriteEvent **with** w
    w.var := v v.val := v
do-forall w: WriteEvent: w.previous?(Self)
    w.previous?(v) := true
do-forall w: WriteEvent: w.overwritten?(Self)
    w.overwritten?(v) := true
do-forall w: WriteEvent: w.knownFrozen?(Self)
    w.knownFrozen?(v) := true

```
rule Execute CRF instruction:
let inst = Self.CRFInsts.head
    case inst.type of
        Freeze:     if Self.cacheValue(inst.addr = inst.val) and Self.cacheStatus(inst.addr)= Clean then
                        Self.cacheValue(inst.addr) := inst.val
                        Self.cacheStatus(inst.addr) := Frozen
                        Proceed
                    else
                        Proceed
                    endif
```

Figure 20: JMM$_{CRF}$: Rule for execution of CRF instructions.

```
rule Add Java instruction:
let inst = Self.currJavaInst
    case inst.type of

    Load:       Add CRF instructions Reconcile(inst.var), LoadI(inst.var), Freeze(inst.var)

    Store:      Add CRF instructions StoreI(inst.var,inst.val), Commit(inst.var), Freeze(inst.var)
```

Figure 21: JMM$_{CRF}$: Rules for adding CRF instructions for Final Fields.

8 Conclusion

We have presented formal semantics of the two proposed replacements for the Java memory model, as well as a variant of Location Consistency appropriate to Java, using the common formalism of Abstract State Machines. Using a single specification technique allows us to compare the models easily. Since the original descriptions of the models are so different, in devising a unified formalization we run the risk of what Börger calls the "formal system straitjacket" [4]: by forcing the specifications to conform to a rigid format, we may lose the essence of the original descriptions. However, ASM's descriptive flexibility allows us to keep our specifications close to the originals.
Researchers involved with the Java memory model have recognized the importance of precise specification. Several formalizations of the original Java memory model have appeared [11, 17, 26]. Yang, Gopalakrishnan, and Lindstrom [34, 33] give specifications of the new Java memory models in terms of their Uniform Memory Model (UMM) framework, based on an operational approach similar to ASM. The authors demonstrate the benefits of formal specification by discovering several mistakes in the original description of the Manson-Pugh proposal. This work is designed for automated verification, and as a result the translation from the original descriptions to UMM is rather involved. For instance, the Manson-Pugh Java memory model is expressed in terms closer to our version of Location Consistency. Integrating our high-level specifications with a lower-level framework like UMM seems an interesting direction for future research.

Predicting the behavior of multithreaded Java applications on various platforms is difficult; even experts can fail to catch subtle but important errors [10, 19]. For Java programmers using multithreading, the ability to simulate executions on different architectures would be a great advantage. We provide executable versions of our ASM specifications, using the XASM tool [32]. Of course, this work ignores the details of Java statements and expressions, as well as important issues like class loading and object initialization. There exist executable ASM specifications of Java that include such details [29, 21], but they do not deal with the underlying memory model. Our next goal is to provide a complete specification of Java that integrates the memory model with other features of the language.

References

[1] Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. Research Report 95/7, Digital Western Research Laboratory (1995)

[2] ASM Home Page. `http://www.eecs.umich.edu/gasm/`

5A Unified Formal Specification and Analysis of the New Java Memory Models.

[3] Blumofe, R.D., Frigo, M., Joerg, C.F., Leiserson, C.E., Randall, K.H.: An Analysis of DAG-consistent Distributed Shared-Memory Algorithms. Proc. ACM SPAA", (1996) 297–308

[4] Börger, E.: Why Use Evolving Algebras for Hardware and Software Engineering? In: Bartosek, M., Staudek, J., Wiedermann, J. (eds.): SOFSEM '95: 22nd Seminar on Current Trends in Theory and Practice of Informatics. LNCS 1012, Springer-Verlag (1995) 236–271

[5] Börger, E. and Schmid, J.: Composition and Submachine Concepts for Sequential ASMs In: Clote, P., Schwichtenberg, H. (eds.): Computer Science Logic (Proceedings of CSL 2000). LNCS 1862, Springer-Verlag (2000) 41–60

[6] Cenciarelli, P., Knapp, A., Reus, B., Wirsing, M.: ¿From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In: Johnson, M. (ed.): Algebraic Methodology and Software Technology. Springer-Verlag (1997) 75–90

[7] Culler, D.E., Singh, J.P., Gupta, A.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann (1999)

[8] Gao, G.R., Sarkar, V.: Location Consistency — A New Memory Model and Cache Consistency Protocol. IEEE Trans. on Comp. 49(8) (2000) 798–813

[9] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. Proc. ISCA (1990) 15–26

[10] Goetz, B.: Double-Checked Locking: Clever, but Broken. JavaWorld 6(2) (2001)

[11] Gontmakher, A., Schuster, A.: Java consistency: Non-operational characterizations for Java memory behavior. ACM Trans. on Comp. Sys. 18(4) (2000) 333–386

[12] Goodman, J.R.: Cache Consistency and Sequential Consistency. Technical Report 1006, Computer Science Dept., U. of Wisconsin–Madison (1989)

[13] Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley (1996)

[14] Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, N., Saphir, W., Snir, M.: MPI: The Complete Reference. MIT Press (1998)

[15] Gurevich, Y.: May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, EECS Department, Univ. of Michigan.

[16] Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In: Börger, E. (ed.): Specification and Validation Methods. Oxford University Press (1995) 9–36

[17] Gurevich, Y., Schulte, W., Wallace, C.: Investigating Java concurrency Using Abstract State Machines. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.): Abstract State Machines: Theory and Applications. LNCS 1912. Springer-Verlag (2000) 151–176

[18] Hagersten, E., Landin, A., Haridi, S.: DDM — A Cache Only Memory Architecture. IEEE Computer 25(9) (1992) 44–54

[19] Holub, A.: Warning! Threading in a Multiprocessor World. JavaWorld 6(2) (2001)

[20] Joe, T.: COMA-F: A Non-Hierarchical Cache Only Memory Architecture. Ph.D. Thesis, Stanford Univ. (1995)

[21] Kutter, P.W.: Montages — Engineering of Programming Languages. Ph.D. Thesis, Eidgenössische Technische Hochschule Zürich (2002)

[22] Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers C-28(9) (1979) 690–691

[23] Maessen, J.-W., Arvind, Shen, X.: Improving the Java Memory Model Using CRF. Proc. OOPSLA (2000) 1–12

[24] Manson, J., Pugh, W.: Multithreaded Semantics for Java. CS Technical Report 4215, Univ. of Maryland (2001)

[25] Pugh, W.: Fixing the Java Memory Model. Proc. ACM Java Grande (1999)

[26] Roychoudhury, A., Mitra, T.: Specifying Multithreaded Java Semantics for Program Verification. Proc. ICSE (2002)

[27] Shen, X., Arvind: Specification of Memory Models and Design of Provably Correct Cache Coherent Protocols. CSG Memo 398, Laboratory for Computer Science, MIT (1997)

[28] Shen, X., Arvind, Rudolph, L.: Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. Proc. ISCA (1999) 150–161

[29] Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer-Verlag (2001)

[30] Wallace, C., Tremblay, G., Amaral, J.N.: On the Tamability of the Location Consistency Memory Model. Proc. PDPTA (2002)

[31] Wallace, C., Tremblay, G., Amaral, J.N.: An Abstract State Machine Specification and Verification of the Location Consistency Memory Model and Cache Protocol. J. Universal Computer Science 7(11) (2001) 1088–1112

[32] XASM Home Page. `http://www.first.gmd.de/~ma/xasm/`

[33] Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Formalizing the Java Memory Model for Multithreaded Program Correctness and Optimization. Technical Report UUCS-02-011, Univ. of Utah.

[34] Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Analyzing the CRF Java Memory Model. Proc. Asia-Pacific Software Engineering Conference (2001) 21-28

[35] James Huggins, Charles Wallace: ASM 101: An Abstract State Machine Primer (Draft) (2002)

[36] Yuri Gurevich: Evolving Algebras : An Attempt to Discover Semantics Electrical Engineering and Computer Science Department The University of Michigan, Ann Arbor

A Introduction to Abstract State Machines (ASMs)

ASMs are a formal operational semantics methodology for documenting software. The advantage of using an ASM specification over a natural language description is that it removes the ambiguity found in natural languages and has a precise meaning. The advantage of an ASM specification over a regular programming language is that a complete ASM can be written at a high level of abstraction, which may not be possible using a programming language without making decisions about the design of the system. Hence an ASM specification can be written much earlier in the software development cycle. ASMs can document a system at various levels of abstraction, and one can write more than one ASM at various stages of development. Because of the formal basis of ASM, it is also a very useful tool for verification.

An ASM is very similar to a computer program, consisting of rules which are equivalent to statements in a normal program. However, in an ASM all the subrules of a program are executed in parallel. Hence in an ASM program we have a parallel logic as against sequential logic in a normal program.

An ASM for a system has two parts to it: the *state*, and the program or *rule* that describes how the state can be changed. An ASM state includes a *superuniverse* : the set of all data values. *Universes* are subsets of the superuniverse and consist of different kinds of data. ASM rules operate on these data. Universes can in fact be considered as functions that return true for all universe members and false for all non-members. The state of an ASM is further described by a set of functions, which describe how the elements of the superuniverse are related. Rules make changes to the states. A run of a sequential ASM program can be considered as a sequence of states, where a new state is obtained by simultaneously applying all rules of the program to the current state.

An ASM *term* can be defined as a variable, or a nullary (0-ary) function name or f(t1,t1..tn) where f is an n-ary function name and t1,t2..tn are terms. Every term evaluates to an element of the superuniverse.

Some of the basic types for ASMs are described below:
1. Update Rule.

foo(t1,t2...tn) := t0

When this rule is executed, the value of the function foo(at the given terms t1,t2..tn) is updated to the value of the term t0.

2.Conditional Rule:

if cond **then**
    rule1
else
    rule2
endif

To execute this rule, execute rule1 if cond evaluates to true, else execute rule2.

3. do-inparallel Rule:

do-inparallel
    rule1
    rule2
    .....
    rulen
enddo

When this rule is executed rules 1 through n are executed in parallel.
(The keywords do-inparallel an enddo are often omitted for brevity).

4. Choose Rule:

choose x: U: Px
    rule1(x)
ifnone
    rule2
endchoose

To execute this rule choose x from the Universe U such that Px evaluates to true and then execute rule1 with x set to that value. If there is no x in universe U such that Px is true, then execute rule2.

5. do-forall Rule

do-forall x: U: cond(x)
    rule(x)
enddo

To execute this rule, for every value of x in Universe U for which cond(x) evaluates to true, execute rule for that value of x. All the executions for different values of x are done in parallel.

In the above discussion we considered ASMs for sequential programs. However we can also design ASMs for distributed environments. In this case we have multiple agents executing different rules. Agents are elements of the universe. Functions map each agent to its ASM program. Agents execute their programs concurrently, updating the global state. Fairness is not guaranteed among the agents: one agent may run for a long time, causing starvation of the other agents. Fairness or any other property of concurrent execution, can be established by simple restricting attention to a class of "legal" runs.

B Semantics of the Manson - Pugh Model

The following lists the Java Memory Model rules proposed by Manson and Pugh [24]. We have ommited the rules for precient writes. However the follwing rules are sufficient for comparison of the models.

writeNormal (Write$< v, w, g >$)
    overwritten$_t \cup =$ previous$_t(v)$
    previous$_t$ $+= < v, w, g >$
    allWrites $+= < v, w, g >$

readNormal (Variable v)
    Choose $< v, w, g >$ from allWrites(v) - overwritten$_t(v)$
    Return w

lock (Monitor m)
    previous$_t \cup =$ previous$_m$
    overwritten$_t \cup =$ overwritten$_m$

unlock (Monitor m)
    previous$_m \cup =$ previous$_t$
    overwritten$_m \cup =$ overwritten$_t$

readVolatile (Variable v)
    previous$_t \cup =$ previous$_v$
    overwritten$_t \cup =$ overwritten$_v$
    return volatileValue$_v$

writeVolatile (Write$< v, w, g >$)
    volatileValue$_v = w$
    previous$_v \cup =$ previous$_t$
    overwritten$_m \cup =$ overwritten$_t$

writefinal (Write$< v, w, g >$)
    finalValue$_v = w$

freezeFinal (Variable v)
    finalValue$_v = w$
    overwritten$_v$ = overwritten$_t$
    knownFrozen$_t$+= v

readFinal (Local$< a, oF, kF >$, Element e)
Let v be the final variable reference by $a.e$
if $v \in kF$
        oF' = overwritten$_v$
    return $< finalValue_v, kF, oF >$ $\cup$ overwritten$_v$
else
    w = either finalValue$_v$ or defaultValue$_v$
    return$< w, kF, oF >$

B.1 Full semantics of the Manson - Pugh Model

For the interested reader we list the full semantics by Manson and Pugh [24].

updateReference(Value w, knownFrozen kf)
    if w is primitive return w
    let$[r, k]= w$
    return$[r, k\cup$ kF]

initWrite(Write$< v, w, g >$)
    $w' =$ updateReference(w,knownFrozen$_t$)
    allWrites $+= < v, w', g >$
    uncommitted$_t$ $+= < v, w', g >$

performWrite(Write$< v, w, g >$)
    $w' =$ updateReference(w,knownFrozen$_t$)
    Assert $< v, w', g > \notin$ previousReads$_t$
    overwritten$_t$ $\cup =$ previous$_t(v)$
    previous$_t$ $+= < v, w, g >$
    uncommitted$_t$ $-= < v, w', g >$

readNormal(Local$< a$,oF,kF>)
    Let v be the variable referenced by $a.e$
    Choose $< v, w, g >$ from allWrites(v) -oF
        -uncommitted$_t$ - overwritten$_t$
    previousReads$_t$ $+= < v, w, g >$
    $< r$,kF$' >=$updateReference(w,knownFrozen$_t$)
    return $< r$,kF$'$,oF>

guaranteedReadOfWrite(Value $< a$, oF,kF>,
    Element e, GUID g)
    Let v be the variable referenced by $a.e$
    Assert $\exists <$ v,w',g $>\in$ previous$_t$
        -uncommitted$_t$ - overwritten$_t$
    previousReads$_t$ $+= < v, w, g >$
    $< r$,kF$' >$ = updateReference(w,knownFrozen$_t$)
    return $< r$,kF$'$,oF>

guaranteedRedundantRead(Value $< a$, oF,kF>,
    Element e, GUID g)
    Let v be the variable referenced by $a.e$
    Let $<$ v,w,g'> be the write seen by g
    Assert $<$ v,w',g $>\in$ previousReads$_t$
        -uncommitted$_t$ - overwritten$_t$
    previousReads$_t$ $+= < v, w, g >$
    $< r$,kF$' >$ = updateReference(w,knownFrozen$_t$)
    return $< r$,kF$'$,oF>

readStatic(Variable v)
    Choose $< v, w, g >$ from allWrites(v)
        -uncommitted$_t$ - overwritten$_t$
    previousReads$_t$ += $< v, w, g >$
    $< r$,kF$' >$ = updateReference(w,knownFrozen$_t$)
    return $< r, \phi$,kF'>

lock (Monitor m)
    Acquire/increment lock on m
    info$_t$ $\cup$ =info$_m$

unlock (Monitor m)
    info$_m$ $\cup$ =info$_t$
    Release/decrement lock on m

readVolatile(Local< a,oF,kf>,Element e)
    Let v be the volatile referenced by $a.e$
    if uncommittedVolatileValue$_v$ $\neq n/a$) or
        (readThisVolatile$_{t,<w,info_t>}$ = false)
        info$_t$ $\cup$ = info$_v$
        return <volatileValue$_v$,kF,oF>
    else
        <w,info$_u$ > = uncommittedVolatileValue$_v$
        volatileValue$_v$=w
        info$_v$ $\cup$ = info$_u$

initVolatileWrite(Write< $v, w, g >$)
    Assert uncommittedVolatileValue$_v$ $\neq n/a$)
    $\forall t \in threads$
        readThisVolatile$_{t,<w,info_t>}$ = false
    uncommittedVolatileValue$_v$ = $< w$, info$_t$ >

performVolatileWrite(Write< $v, w, g >$)
    uncommittedVolatileValue$_v$ = n/a) or
        volatileValue$_v$=w
        info$_v$ $\cup$ = info$_u$

writefinal (Write< $v, w, g >$)
    finalValue$_v$ = w

freezeFinal (Variable v)
    finalValue$_v$ = w
    overwritten$_v$ = overwritten$_t$
    knownFrozen$_t$+= v

C Semantics of the CRF Model

The following lists the semantics for the CRF Model [23].

Local Rules:
$(r = Storel(a, v); instr, comp, cache[a := -, -]) \Rightarrow (instr, r = \surd/comp, cache[a := v, Dirty])$

$(r = Loadl(a); instr, comp, cache[a := v, s]) \Rightarrow (instr, r = v/comp, cache[a := v, s])$

$(r = Commit(a, v); instr, comp, cache) \Rightarrow (instr, r = \surd/comp, cache)$
where a is not in $cache$ or a is $Clean$

$(r = Reconcile(a, v); instr, comp, cache) \Rightarrow (instr, r = \surd/comp, cache)$
where a is not in $cache$ or a is $Dirty$

$(r = Fence(a, b); instr, comp, cache) \Rightarrow (instr, r = \surd/comp, cache)$

$(r = Freeze(a); instr, comp, cache[a := v, Clean]) \Rightarrow (instr, r = v/comp, cache[a := v, Frozen])$

$(r = Freeze(a, v); instr, comp, cache) \Rightarrow (instr, r = \surd/comp, cache)$

Background Rules:

$(instr, comp, cache[a := v, s]) \Rightarrow (instr, comp, cache)$
where s is $Clean$ (Eject a cache entry)

$(instr, comp, cache)/threads, memory[a := v] \Rightarrow (instr, comp, cache[a := v, Clean])/threads, memory[a := v])$
where $cache$ contains no mapping for a (Fetch a value from main memory)

$(instr, comp, cache[a := v, Dirty])/threads, memory[a := -] \Rightarrow$
$(instr, comp, cache[a := v, Clean])/threads, memory[a := v]$ (Write back a value to main memory)

Local Rules:

$(r = Lock(l); instr, comp, cache[l := n, Locked]) \Rightarrow (instr, r = n/comp, cache[l := n + 1, Locked])$

$(r = Unlock(l); instr, comp, cache[l := n + 1, Locked]) \Rightarrow (instr, r = n/comp, cache[l := n, Locked])$

Background Rules:

$(instr, comp, cache)/threads, memory[l := 0] \Rightarrow (instr, comp, cache[l := 0, Locked])/threads, memory[a := v])$

$(instr, comp, cache)/threads, memory[l := 0, Locked] \Rightarrow (instr, comp, cache)/threads, memory[l := 0])$