Investigating Java Concurrency Using Abstract State Machines

Yuri Gurevich[1], Wolfram Schulte[1], and Charles Wallace[2]

[1] Microsoft Research, Redmond, WA 98052-6399, USA
{gurevich,schulte}@microsoft.com
[2] University of Delaware, Newark, DE 19716, USA
wallace@cis.udel.edu

Abstract. We present a mathematically precise, platform-independent model of Java concurrency using the Abstract State Machine method. We cover all aspects of Java threads and synchronization, gradually adding details to the model in a series of steps. We motivate and explain each concurrency feature, and point out subtleties, inconsistencies and ambiguities in the official, informal Java specification.

1 Introduction

The *Java* programming language [7, 13] provides sophisticated support for concurrency. The fundamental operations of concurrent programs are implemented as built-in features of the language. Many of the notoriously complicated details of concurrent programming are hidden from the programmer, simplifying the design of concurrent applications. Furthermore, a platform-neutral memory model is included as part of the language. The incorporation of such intricate, subtle operations into Java calls for a precise specification. As interest in the language's concurrency model increases, Java developers are examining and exploiting its details [15, 17, 16, 14]. The popularity of Java and, more importantly, its emphasis on cross-platform compatibility make the need for such a specification even stronger.

We present a model of the concurrent features of Java, using the formal operational specification method of *Abstract State Machines (ASMs)*[1] [9, 22, 23]. We use the *Java Language Specification* manual *(JLS)* [7], as our reference for the language. The JLS is an informal specification, and due to the ambiguity which pervades natural language, it can be interpreted in different ways. Our model gives an unambiguous specification which reflects our interpretation of the JLS. Throughout the paper, we indicate where ambiguities and omissions in the JLS give rise to other interpretations. The formal specification process also uncovers many subtle but important issues which the JLS does not bring to light. Our goal is a specification that is not only precise but accessible to its readers, even those not familiar with Java, concurrent programming, or ASMs. As part of this

[1] ASMs were formerly known as *Evolving Algebras*.

project, we implement the ASM specifications using the *ASMGofer* interpreter [20]. This implementation serves as a convenient tool for prototyping and testing.

It is important to distinguish unintentional ambiguity from intentional underspecification. The authors of the JLS envision support for Java concurrency taking many different forms: "by having many hardware processors, by time-slicing a single hardware processor, or by time-slicing many hardware processors [7]". The JLS leaves some details unspecified in order to give implementers of Java a certain amount of freedom: "the intent is to permit certain standard hardware and software techniques that can greatly improve the speed and efficiency of concurrent code [7]". As is usual with the ASM methodology [8, 2], we design our ASMs to model Java concurrency at its natural level of abstraction, capturing the essence of the JLS without committing to any implementation decisions.

While most of the JLS specification is imperative in nature, some of the concurrency model is described in a declarative style: rather than explain how to execute a concurrent program, the JLS gives conditions that a correct execution must meet. This shift in presentation style is most likely due to the notion that only a declarative specification can be truly implementation-independent.

There are two ways for us to deal with the declarative portions of the JLS. The obvious and simple way is to reformulate them declaratively as conditions on the execution of our ASM. The other way is to implement them in the ASM itself. We choose to do the latter, whenever it is natural to do so, but in a way that does not sacrifice generality. Our model obeys the conditions established in the JLS, yet it is fully general in the sense that for any concurrent program execution that follows the rules of the JLS, there is a corresponding run of the ASM. We find our imperative approach helpful in creating a clear mental picture of the concurrency model, and in uncovering hidden assumptions, inconsistencies and ambiguities in the JLS.

There are several formalizations of Java concurrency in the literature. Börger and Schulte [3] also uses ASMs, to give a semantic analysis of Java that exposes a hierarchy of natural sublanguages. It covers the language features for concurrency, but its focus is on the high-level language rather than the lower-level concurrency model, so it does not provide a full specification of Java concurrency. There are several works [1, 4, 5] using the *Structural Operational Semantics (SOS)* methodology [18]. Attali *et al.* [1] do not concentrate on the lower-level details of Java concurrency, while Cenciarelli *et al.* [4] give them in a declarative style, in keeping with the presentation in the JLS. As mentioned earlier, we believe that there are advantages to our imperative approach. Coscia and Reggio [5] explicitly deal with the details of the concurrency model and propose some changes to its design. Gontmakher and Schuster [6] present a non-imperative specification, with the purpose of comparing Java's memory behavior with other well-known notions of consistency. Our focus is different, however: while their work assumes a particular interpretation of the JLS and proceeds from there, our goal is first to come to a clear understanding of the concurrency model as

presented in the JLS, and then to discuss its consequences. As we shall see, there are several issues within the JLS which make this initial understanding difficult.

We introduce in §2 the basic rules by which agents (*threads*) interact with regard to shared *variables*, and in §3 the special considerations for variables marked as *volatile*. In §4 we introduce *locks*, which are used to limit concurrency within a program. In §5 we discuss *prescient* store actions. In §6 we describe *thread objects*. In §7, we cover *waiting* and *notification*.

Our technical report [11] provides a complete presentation of our ASM specification. In it, we also prove that our specification is as general as possible while obeying all the correctness criteria of the JLS.

Acknowledgment. This work was partially supported by NSF grant CCR-95-04375.

2 Threads and variables

In a *single-threaded* computation, a single agent executes the instructions of a program, one at a time. In contrast, a run of a Java program may be *multi-threaded*, (intuitively) involving multiple agents that execute instructions concurrently. Each agent executes sequentially, the sequence of its actions forming a *thread* of execution. In the parlance of distributed computing, the term "thread" is used to refer not only to an agent's computation but also to the agent itself. Threads may execute on a single processor or multiple processors. The atomic actions of different threads may be interleaved; they may even be concurrent in the case of multiple processors.

Since different threads may access and update common data, the order of their actions affects the results of an execution. The JLS establishes some conditions on the interaction of threads with respect to shared data, but intentionally leaves some freedom to implementers of the language. Consequently, the results of executing a multithreaded program may vary between different Java platforms, and even between different executions on the same platform.

A data value is either an instance of a primitive type, such as `int` or `boolean`, or an *object*, a dynamically created value. A value resides at a location either in the *working memory* of a thread or in the *main memory*. Certain memory locations serve as manifestations of a *variable*. The *master copy* of a variable is the unique location in the main memory that manifests the variable. A thread's *working copy* of a variable is the unique location in the thread's working memory that manifests the variable. A thread accesses or updates only its working copies.

The JLS speaks of the main memory as an agent which communicates with threads and updates variable master values. While this view is adequate for describing a system with a centralized main memory, it does not capture the possibility of a distributed main memory [16], where updates of different variable master copies may occur concurrently. We find it convenient to imagine that each variable has a *master* agent, which accesses and updates the variable's master copy. Since there is a one-to-one correspondence between variables and their masters, we may identify a variable master with the variable it controls.

We create an ASM to model threads' actions on variables. The agents of are threads (members of Thread) and variable masters (members of Var). At the beginning of each run, there is a single member of the universe Thread. Since we identify variable masters with the variables they control, a member of the universe Var is both a variable and a variable master. The universe Value represents the set of values that master or working copies of variables may bear. The function masterValue maps each variable to the current value of its master copy. The function workingValue, given a thread and variable, returns the current value of the thread's working copy of that variable.

A thread's *execution engine* is the component of the thread that executes the Java program. It may perform actions on variables: *creating* a variable, *assigning* a new value to a variable, or *using* a previously assigned value of a variable. There may be more than one value associated with a variable, but a thread has immediate access only to the value of its working copy. A use or assign action is internal to a thread, involving only its working copy. Since we are not interested here in what a thread computes during its execution, our view of use and assign actions is simple: an assign action just changes the value of a variable's working copy, and a use action does nothing. (Although what happens in a use action is not important to us, we shall see in §3 that the *occurrence* of a use action may be significant, even at our level of abstraction.) A thread's execution engine may do one other sort of action of interest to us: creating another thread, which may execute concurrently with its creator.

Threads pass values of shared variables among themselves via the main memory. A thread may update the master copy of a variable with a freshly assigned value, or it may request the current value of the master copy. This is done through asynchronous communication with a variable's master agent. To transfer the value of its master copy to a thread's working memory, (the master of) a variable issues a *read* action. This is followed by a *load* action by the thread, which installs the value into its working memory. To transfer its working value of a variable to the main memory, a thread issues a *store* action. This is followed by a *write* action by (the master of) the variable, which installs the value into the main memory.

Somehow information is passed from variable to thread in a read-load sequence, and from thread to variable in a store-write sequence. To give a more imperative character to our description, without any essential loss of generality, we introduce an explicit but quite abstract notion of *messages* passing between threads and variables. When a variable issues a read action, it sends an *access message* to a thread, and the target thread then receives the message by issuing a load action. When a thread performs a store action, it sends an *update message* to a variable, and the variable then receives the message by issuing a write action. So for example, we may speak of a thread storing out to a variable by sending it an access message, or a variable writing in from a thread by installing the value of an update message.

We define a universe Msg comprising the universe AccMsg (messages from variables to threads) and the universe UpdMsg (messages from threads to vari-

ables). The function var maps each access message to the variable that sent it, and maps each update message to the variable that is its intended recipient. The function thread maps each update message to the thread that sent it, and maps each access message to the thread that is its intended recipient. The function value returns the value contained in a given message.

In any state, the members of the universe AccMsg that have been sent from a variable v to a thread t form a "subuniverse" of AccMsg, which we call AccMsg(v, t). Likewise, the members of the universe UpdMsg that have been sent from t to v form a subuniverse of UpdMsg, which we call UpdMsg(t, v).

While the JLS does not dictate a specific policy on the access of shared variables, it does impose some rules on the concurrent behavior of threads and variables. We present these rules as they appear in the JLS and give our interpretations of them.

JLS rules 1

1. [p. 403] "The actions performed by any one thread are totally ordered; that is, for any two actions performed by a thread, one action precedes the other."
2. [p. 403] "The actions performed by the main memory for any one variable are totally ordered; that is, for any two actions performed by the main memory on the same variable, one action precedes the other."
3. [p. 403] "It is not permitted for an action to follow itself."

In other words, actions on variables form a partial order (which we denote by $<$) in which the actions of any single thread are linearly ordered, and the actions of any single variable master are linearly ordered. Note that this supports our view of independent variable master agents as opposed to a monolithic main memory. The read and write actions on a single variable are ordered linearly, but read and write actions on different variables may occur concurrently. This follows naturally if we think of the master copy of each variable as controlled by an independent agent.

JLS rules 2

1. [p. 403] "Each load action by a thread is uniquely paired with a read action by the main memory such that the load action follows the read action."
2. [p. 403] "Each store action by a thread is uniquely paired with a write action by the main memory such that the write action follows the store action."
3. [p. 405] "For every load action performed by any thread t on its working copy of a variable v, there must be a corresponding preceding read action by the main memory on the master copy of v, and the load action must put into the working copy the data transmitted by the corresponding read action."
4. [p. 405] "For every store action performed by any thread t on its working copy of a variable v, there must be a corresponding following write action by the main memory on the master copy of v, and the write action must put into the master copy the data transmitted by the corresponding store action."

5. [p. 405] "Let action A be a load or store by thread t on variable v, and let action P be the corresponding read or write by the main memory on variable v. Similarly, let action B be some other load or store by thread t on that same variable v, and let action Q be the corresponding read or write by the main memory on variable v. If A precedes B, then P must precede Q."

These rules restrict the ways in which a thread's load and store actions may be interleaved with a variable's read and write actions. First, we note that the term "uniquely paired" in Rules 2.1 and 2.2 is ambiguous. A statement of the form "Each element x of X is uniquely paired with an element y of Y such that $\phi(x, y)$" has a lenient interpretation: "For all x in X there is a unique y in Y such that $\phi(x, y)$". It also has a strict interpretation: "For all x in X there is a unique y in Y such that $\phi(x, y)$, and for all y in Y there is a unique x in X such that $\phi(x, y)$". The lenient interpretation allows *spurious* y's; that is, y's that are not paired with any x. Which interpretation shall we use? The strict interpretation is what the authors of the JLS intended [21], so we adopt it here. However, we find that a lenient interpretation of the term in Rule 2.1 has some advantages that make it worth discussing.

Read/load order. For every load action L by a thread t on a variable v, there is a read action $R < L$ by v such that L loads in the value read out at R (Rules 2.1 and 2.3). This supports our conception of access messages: every load action must load in the value of an access message issued by a previous read action. Furthermore, a thread may not load in the same access message twice (Rule 2.5).

The choice of interpretation of "uniquely paired" determines whether every access message must be loaded in. The strict interpretation, under which every access message is loaded in, is simpler and more appealing from a logical perspective, which is why it is the intended interpretation [21]. But note that spurious read actions are innocuous: failing to load in a given access message has only the effect of making a thread's working memory less up-to-date than possible. Furthermore, the lenient interpretation allows a higher degree of independence of threads and variables: a variable is free to issue access messages without concern for whether they are eventually loaded in. For instance, updated master values of an important variable could be broadcast to threads, with each thread deciding whether to load the value in; or blocks (*pages*) of master values could be sent to a thread, with the thread selecting which values to load in.

Store/write order. For every store action S by a thread t on a variable v, there is a write action $W > S$ by v such that W writes in the value stored out at S (Rules 2.2 and 2.4). Using our message-passing parlance, every store action sends an update message which must be written in by a following write action.

As with the case of read and load actions, how we interpret "uniquely paired" determines whether every write action is an action of writing in an update message. Here, adopting the strict interpretation seems less controversial. It is not clear what value a spurious write action writes. At best, it simply rewrites the value that is already in main memory, in which case it is useless. But if it writes

a different value, it may obliterate a current value in main memory, replacing it with an outdated or invalid value.

Read/write order follows load/store order. Each thread performs load and store actions on a variable in a certain order, and the variable must perform the corresponding read and write actions in the same order. Let us examine this more closely.

Load/load. If a thread t performs a load action L_1 (with corresponding read action R_1) on a variable v, and another load action L_2 (with corresponding read action R_2) on v, Rule 2.5 postulates that $(L_1 < L_2) \Rightarrow (R_1 < R_2)$. If v sends access messages m and m' to t, then t may load in m before m' only if m was sent before m'. In other words, t loads in ever newer access messages from v. Without loss of generality, we may think of t as discarding m when it loads it in.

The conjunction of Rule 2.3 and Rule 2.5 implies Rule 2.1. Rule 2.1 asserts that each load action has a preceding unique read action. Rule 2.3 asserts that every load action has a preceding read action and Rule 2.5 asserts that different load actions must have different corresponding read actions, ensuring uniqueness.

Store/store. If a thread t performs a store action S_1 on v (with corresponding write action W_1) and another store action S_2 on v (with corresponding write action W_2), Rule 2.5 postulates that $(S_1 < S_2) \Rightarrow (W_1 < W_2)$. If t sends an update message m to v before sending another update message m' to v, then m must be written before m'. In other words, v writes in ever newer values stored out by t. Thus writing m' is allowed only if m has been delivered through a write action. We can think of v as discarding m when it writes it.

Store/load. If a thread t performs a store action S (with corresponding write action W) on v, and t performs a load action L (with corresponding read action R) on v, Rule 2.5 postulates that $(S < L) \Rightarrow (W < R)$. In other words, if t sends an update message m to v before loading an access message m' in from v, then v must write m in before sending m'. If t stores its working value out to v and then immediately loads in from v, and other threads do not store out to v in the meantime, then the value it receives from the load is the value it stored.

Thus sending m' is allowed only *after* the point in time when m is written in. If m' is sent but not loaded in *before* this point in time, it can never be loaded in, and thus Rule 2.1 is violated (under the strict interpretation). To avoid this, v must not issue an access message to t as long as there is a pending update message from t to v, and t must not issue an update message to v as long as there is a pending access message from v to t. Furthermore, v and t must be prevented from sending messages to each other concurrently.

Under the assumption that messages take no time to travel, we may require that v checks for incoming update messages from t before sending an access message to t, and t checks for incoming access messages from v before sending an update message to v. But this does not eliminate the possibility of concurrent actions by v and t. Thus some means of avoiding concurrent actions by v and t is necessary. Note that this complication disappears if we were to adopt the lenient

interpretation of "uniquely paired" for the pairing of read and load actions. In our example, t could simply ignore the access message m' and still have its update message m written.

To represent the relative age of messages, we introduce the relation $<$, and the term oldest? to determine whether a given message has no predecessors.

term oldest?(m): (not $\exists m'$: Msg) $m' < m$

The JLS does not impose any particular means of ordering messages, so we avoid doing so here by making the function external. We restrict attention to runs in which the function $<$ behaves in the expected way.

JLS rules 3 Let t be a thread and v be a variable.

1. [p. 404] "A use or assign action by t of v is permitted only when dictated by execution by t of the Java program according to the standard Java execution model."
2. [p. 404] "A store action by t on v must intervene between an assign by t of v and a subsequent load by t of v."
3. [p. 404] "An assign action by t on v must intervene between a load or store by t of v and a subsequent store by t of v."
4. [p. 404] "After a thread is created, it must perform an assign or load action on a variable before performing a use or store action on that variable."
5. [p. 405] "After a variable is created, every thread must perform an assign or load action on that variable before performing a use or store action on that variable."

These rules impose constraints on the exchange between a thread's execution engine and working memory.

Use and assign actions. Threads may only issue use and assign actions when needed for progress by the execution engine (Rule 3.1). At the level of abstraction we have chosen, we are not concerned with what threads actually compute, so this rule does not affect our model. Nevertheless, we should note that the term "dictated" is somewhat ambiguous here. For example, let `var` be an expression that requires a use action on a variable v. In evaluating the expression `(var + var)`, how many use actions on v are dictated by the program? The strictest interpretation would require a use action for each reference to a variable (our example would require two use actions), and this is what the authors of the JLS intended [21].

Store actions. If a thread t performs a load or store action LS on v and then performs a store action S on v, Rule 3.3 postulates that t performs an assign action A on v such that $LS < A < S$. In other words, spurious store actions are forbidden: t may only store out to v if it has a new value to transmit. This prevents the current contents of the main memory from being overwritten by older (*i.e.*, previously stored) values. Without this rule, a thread could store out outdated values it loaded in long ago, overwriting more current values that other threads stored out in the interim.

If a thread t performs an assign action A on a variable v and then performs a load action L on v, there is a store action S by t on v such that $A < S < L$ (Rule 3.2). In other words, a thread must store out a value of a variable v after issuing a sequence of assigns to v, transmitting at least the last assigned value to the main memory, before attempting a load; the assigns are not completely forgotten. A thread may load an access message in from a given variable only if the last working value it assigned has been stored out. Note that a thread may assign to a variable and then use the variable with no intervening load. In this case, no intervening store is required, and the assigned value may simply remain in the working memory.

This makes the coordination between threads and variables more complex. Once t performs an assign on v, its next action on v cannot be a load. So if t assigns a value to v concurrently with v sending an access message to t, t may not subsequently load the message in until it stores its working value out to v. But it cannot perform this store action, as the write action corresponding to the store would follow v's read action; thus Rules 2.1 and 2.3 are violated. To avoid this, v must not issue an access message to t if there is an assign action by t on v that has not been followed by a store action on v, and t must not assign to v if there is a pending access message. So the actions which threads and variables must avoid doing concurrently are not reads and stores, as Rule 2.5 might suggest, but reads and assigns.

Use/store actions preceded by an assign/load action. If a thread t performs a use or store action US on a variable v, Rules 3.4 and 3.5 postulate that there is an assign or load action $AL < US$ by t on v. For use and store actions to behave properly, there must be a valid value in working memory. Initially, all the contents of a thread's working memory are invalid, but as a result of these rules, no invalid value of the working memory is ever used or stored out.

We define the following functions to help enforce the conditions described above. Rule 3.3 allows a thread t to store out its working value of a variable v only if it has assigned a fresh value to v but has not stored it out; Rule 3.2 allows t to load an access message in from v only if it has no freshly assigned working value of v to store out. The function freshAssign? determines whether a given thread has assigned a fresh value to a given variable without storing it out. Rules 3.4–5 allow t to use its working value of v only if there is a valid value of v in its working memory, installed there by a load or assign action. The function usableValue? determines whether a given thread has a valid value for a given variable in its working memory.

The actions of a variable master consist of issuing a read message to a thread, and writing in an update message.

module Var: **choose among**
    choose t: Thread: readOK?(t)
        Read out to t
    choose t: Thread
        choose m: UpdMsg(t, Self): writeOK?(m, t)
            Write m in from t

rule *Read out to t*: **extend** AccMsg(Self, t) **with** m
value(m) := masterValue(Self)

rule *Write m in from t*: masterValue(Self) := value(m)
UpdMsg(m) := false

The terms readOK? and writeOK? determine whether the given action is allowed by the JLS rules. Rules 2.5 and 3.2 allow a variable v to read out to a thread t only if every assign action by t on v has been followed by corresponding store and write actions. Rule 2.5 allows t to write a message in from v only if the message is the oldest pending message from t to v.

term readOK?(t): not (freshAssign?(t, Self) or ($\exists m$: UpdMsg(t, Self)))

term writeOK?(m, t): oldest?(m)

The actions of a thread consist of storing a value out through an update message, loading in an access message, or taking a step in executing the Java program. Program execution may involve using the working value of a variable, assigning a value to the working copy of a variable, creating a variable, or creating a thread.

module Thread: **choose among**
Execute program *WM-MM transfer*

rule *Execute program*: **choose among**
WM-EE transfer *Create var* *Create thread*

rule *WM-EE transfer*: **choose among**
choose v: Var: useOK?(v)
Use v
choose v: Var: assignOK?(v)
Assign to v

rule *WM-MM transfer*: **choose among**
choose v: Var
choose m: AccMsg(v, Self): loadOK?(m, v)
Load m in from v
choose v: Var: storeOK?(v)
Store out to v

We define rules for load, store, use and assign actions, as well as the actions of creating a variable and creating a thread.

rule *Load m in from v*: workingValue(Self, v) := value(m)
usableValue?(Self, v) := true
AccMsg(m) := false

rule *Store out to v*: freshAssign?(Self, v) := false
extend UpdMsg(Self, v) **with** m
value(m) := workingValue(Self, v)

rule *Use* v: **skip**

rule *Assign to* v: freshAssign?(Self, v) := true
usableValue?(Self, v) := true
choose val: Value
workingValue(Self, v) := val

rule *Create var*: **create** Var **agent** v

rule *Create thread*: **create** Thread **agent** t

The terms loadOK?, storeOK?, useOK? and assignOK? determine whether the given action is allowed by the JLS rules. Rule 2.5 allows a thread t to load a message in from a variable v only if the message is the oldest pending message from v to t. Rules 3.3–5 allow t to store out to v only if there has been an assign action by t on v without a following store action. Rules 3.4–5 allow t to use v only if there has been an assign or load action that put a value in t's working copy of v. Rules 2.5 and 3.2 allow t to assign to v only if every access message from v to t has been loaded in.

term loadOK?(m, v): oldest?(m) and not freshAssign?(Self, v)

term storeOK?(v): freshAssign?(Self, v)

term useOK?(v): usableValue?(Self, v)

term assignOK?(v): (not $\exists m$: AccMsg(v, Self))

3 Volatile variables

The basic model for threads and shared variables, as presented in §2, permits optimizations that reduce the amount of communication between agents and thus enhance the performance of multithreaded programs.

Communication between threads and the main memory can be lessened through *caching*: keeping values in working memory without transmitting them to main memory or getting new values from main memory. Instead of consulting the main memory every time it uses a variable, a thread may service several use actions on the same variable with a single load action. It may also service several assign actions with a single store action, sending only the last of these assigned values to main memory.

Caching may have undesirable results, particularly for variables that are assigned to and used frequently by different threads. Cached values may become outdated as other threads store out to the main memory. Also, caching assigned values prevents other threads from viewing the newly assigned values as they arise.

Communication between variables can be avoided altogether, allowing them to operate independently of one another. While Rule 2.5 dictates that the order of a thread's actions on a *single* variable is also followed by the variable, the order

of its actions over *different* variables need not be followed in the main memory. Consequently, variables need not coordinate their actions among themselves.

Hence for certain variables, a stricter discipline is necessary. Java allows the programmer to declare variables *volatile* at the time of their creation. Volatile variables follow a policy that disallows the optimizations described above.

We modify the ASM to model operations on volatile variables. We add a universe VolVar, representing the set of volatile variables. Every member of VolVar is also a member of Var. The rule *Create var* requires a slight change: a new variable may be marked as volatile.

The JLS imposes the following additional conditions on volatile variables.

JLS rules 4 [p. 407] "Let t be a thread and let v and w be volatile variables."

1. "A use action by t on v is permitted only if the previous action by t on v was load, and a load action by t on v is permitted only if the next action by t on v is use. The use action is said to be "associated" with the read action that corresponds to the load."
2. "A store action by t on v is permitted only if the previous action by t on v was assign, and an assign action by t on v is permitted only if the next action by t on v is store. The assign action is said to be "associated" with the write action that corresponds to the store."
3. "Let action A be a use or assign by thread t on variable v, let action F be the load or store associated with A, and let action P be the read or write of v that corresponds to F. Similarly, let action B be a use or assign by thread t on variable w, let action G be the load or store associated with B, and let action Q be the read or write of v that corresponds to G. If A precedes B, then P must precede Q."

Caching is disallowed. If a thread t performs a load action L on v, Rule 4.1 postulates that there is a use action $U > L$ by t on v, and there is no action Act by t on v such that $L < Act < U$. In other words, each value loaded in from the main memory is used exactly once. In contrast, multiple use actions on a non-volatile variable may use a value loaded in by a single load action.

Rule 4.2 postulates that a thread t performs a store action S on a volatile variable v if and only if there is an assign action $A < S$ by t on v and there is no action Act by t on v such that $A < Act < S$. In other words, every assigned value must be stored out and written to the main memory exactly once. For non-volatile variables, assign actions may overwrite one another without having their values stored out.

Read/write order follows use/assign order. Each thread performs use and assign actions on volatile variables in a certain order; Rule 4.3 ensures that the variables must perform the corresponding read and write actions in the same order. This condition is similar to Rule 2.5, but different in some important ways. First, the thread actions mentioned in Rule 4.3 are use and assign actions, as opposed to store and load actions as in Rule 2.5. Also, the ordering holds over actions on all volatile variables, not just those on a single variable.

To account for the behavior of volatile variables, we modify our view of message passing between threads and variables. We think of a volatile variable access message as ending in a use action, as opposed to a load action. Similarly, we think of a volatile variable update message as originating with an assign action, as opposed to a store action. By Rules 2.4 and 4.2, all volatile update messages must be stored out and written, and by Rules 2.3 and 4.1, all volatile access messages must be loaded in and used. We examine this more closely.

Use/Use. If a thread t performs a use action U_1 (with corresponding read action R_1) on a volatile variable v, and another use action U_2 (with corresponding read action R_2) on a volatile variable w, Rule 4.3 postulates that $(U_1 < U_2) \Rightarrow (R_1 < R_2)$. Note that R_1 and R_2 may be actions by different variables. Given an access message m from v and an access message m' from w, t may use m before m' only if m was sent before m'. In other words, t uses ever newer volatile access messages.

The conjunction of Rule 4.3 with Rules 4.1 and 3.1 implies that volatile variables must work closely with threads' execution engines. Once a thread loads in a read message from a volatile variable, by Rule 4.1 it must use that variable, and by Rule 4.3 it must do so before performing any other assign or use action. Rule 3.1 forbids use actions not "dictated" by the execution engine, so the message must be sent with knowledge of what is needed by the Java program. Consider a scenario where a thread has just loaded an access message from a volatile variable v, but the next operation by the execution engine is an assign action on v, or an action on another volatile variable. The only way for t to progress is to issue a gratuitous use action on v, but this is forbidden by Rule 3.1. Therefore, volatile variables must be careful when issuing access messages to a thread, doing so only when it is clear that the thread's execution needs one immediately.

Rule 4.3 also implies that volatile variables must not read out to the same thread concurrently. As the actions of a single thread are linearly ordered, by Rule 4.3 the corresponding read actions must be linearly ordered as well. This may require some coordination between volatile variables; if several volatile variables wish to read out to a single thread, only a single variable at a time may do so.

Assign/Assign. If a thread t performs an assign action A_1 (with corresponding write action W_1) on a volatile variable v, and another assign action A_2 (with corresponding write action W_2) on a volatile variable w, Rule 4.3 postulates that $(A_1 < A_2) \Rightarrow (W_1 < W_2)$. Note that W_1 and W_2 may be actions by different variables. In other words, given an update message m from t to v and an update message m' from t to w, v may write m before m' only if m was sent before m'. So volatile update messages sent by t are written in the order in which they are sent.

Assign/Use. If a thread t performs a use action U (with corresponding read action R) on a volatile variable v, and an assign action A (with corresponding write action W) on a volatile variable w, Rule 4.3 postulates that $(A < U) \Rightarrow$

($W < R$). In other words, if t sends a volatile update message m before using a volatile access message m', then m must be written in before m' is sent.

Thus sending m' is allowed only *after* the point in time when m and all other volatile update messages from t have been written. If m' is sent but not used *before* this point in time, it can never be used, hence violating Rule 4.1. To avoid this, v must not issue an access message to t as long as there is a pending volatile update message from t, and t must not issue a volatile update message as long as there is a pending volatile access message to t. Furthermore, volatile access and update messages must not be sent to and from t concurrently.

We define some additional functions to help enforce these conditions. Rule 4.1 requires a thread t to use a volatile variable v if and only if it has loaded in a message from v without using its value. If a thread has loaded in a message from a volatile variable but not used its value, the function msgToUse returns this message. If a thread has assigned a value to a volatile variable but not stored the value out, the function msgToStore returns the volatile update message.

The terms readOK? and writeOK? include extra conditions on volatile variables. Rule 4.3 allows a volatile variable v to read out to a thread t only if all assign actions by t on volatile variables have been followed by corresponding store and write actions. To readOK? we add the conjunct VolVar(Self) $\Rightarrow$ ($\forall v$: VolVar) (not $\exists m$: UpdMsg(t, v)). Rule 4.3 allows v to write an update message from t in only if the message has been stored out and is the oldest pending volatile message from t. To writeOK? we add the conjunct VolVar(Self) $\Rightarrow$ $m \neq$ msgToStore(t, Self).

We modify the rules for load and store actions. A volatile access message is removed when it is used, rather than when it is loaded in. A volatile update message is created through an assign action, rather than through a store action.

rule *Load m in from v*: workingValue(Self, v) := value(m)
usableValue?(Self, v) := true
if VolVar(v) **then** msgToUse(Self, v) := m
else AccMsg(m) := false

rule *Store out to v*: freshAssign?(Self, v) := false
if VolVar(v) **then** msgToStore(Self, v) := undef
else extend UpdMsg(Self, v) **with** m
value(m) := workingValue(Self, v)

The term loadOK? includes an extra condition on volatile variables. Rule 4.1 allows a thread t to load a message in from a volatile variable v only if it has used all previous loaded access messages from v. Rule 4.2 allows t to load a message in from v only if it has stored out all previously assigned values to v. To loadOK? we add the conjunct VolVar(v) $\Rightarrow$ not usableValue?(Self, v).

We modify the rules for use and assign actions. A use action on a volatile variable removes the volatile access message from the AccMsg universe. An assign action on a volatile variable generates a UpdMsg.

```
rule Use v: if VolVar(v) then
                usableValue?(Self, v) := false
                msgToUse(Self, v) := undef
                AccMsg(msgToUse(Self, v)) := false

rule Assign to v: freshAssign?(Self, v) := true
                  if not VolVar(v) then  usableValue?(Self, v) := true
                  choose val: Value
                      workingValue(Self, v) := val
                      if VolVar(v) then
                          extend UpdMsg(Self, v) with m
                              val(m) := val
                              msgToStore(Self, v) := m
```

The terms useOK? and assignOK? include extra conditions on volatile variables. Rule 4.1 allows a thread t to use its working value of a variable v only if it has loaded in a message from v without using its value. Rule 4.3 requires this message to be the oldest pending volatile access message to t. To useOK? we add the conjunct VolVar(v) $\Rightarrow$ oldest?(msgToUse(Self, v)). Rule 4.2 allows t to assign a value to v only if all of t's previous assigns to t have been stored out, and Rule 4.3 requires that there be no pending volatile access messages to t. To assignOK? we add the conjunct VolVar(v) $\Rightarrow$ not (freshAssign?(Self, v) or usableValue?(Self, v) or ($\exists m$: AccMsg(v', Self))).

If two volatile variable variables read out to t concurrently, the corresponding load actions will be ordered, since t can only perform one load action at a time. But the read actions will not be ordered, thereby violating Rule 4.3. We restrict attention to runs in which this does not occur.

Rules 4.1 and 4.2 ensure that if a load or assign action on a VolVar occurs, a corresponding use or store action will occur sometime in the future. Similarly to Rules 2.1 and 2.2, these rules can be flouted by delaying a use or store action indefinitely. We restrict attention to runs that avoid this situation.

4 Locks

Certain situations require that a thread be able to perform a series of operations without interference from other threads. In general, the programs of threads $t_1 \ldots t_n$ may have critical regions that they should avoid executing concurrently. It seems reasonable to try to prevent any t_i and t_j from entering their critical regions simultaneously. A natural way to solve this problem in an object-oriented framework is to have one critical *object* related to all these critical regions. All the threads may refer to the critical object, but only one thread may own the object at any given time, and only the owner can execute its critical region. Java provides support for this sort of mutual exclusion. The critical regions are called *synchronized* regions.

In Java, each object has a unique *lock* associated with it. A thread gains entry into a synchronized code region of its program by performing a *lock* action

on the lock of the critical object. The thread then *holds* the lock until it exits the synchronized code region and performs an *unlock* action on the lock.

Various threads may issue lock or unlock actions on the same lock. The lock and unlock actions on a single lock are performed in conjunction with an arbiter, which restricts the number of threads holding a single lock to one at a time. The JLS speaks of main memory as this arbiter. As with variable master copies, we prefer to think of each lock as controlled by a *master* agent, rather than by the (possibly distributed) main memory.

We modify the ASM to model operations on locks. The agents are threads, variable masters, and lock masters. We represent locks as members of the universe Lock. As with variables, we identify lock master agents with the locks they control, so a member of the universe Lock is both a lock and a lock master. Objects are members of the universe Object. A LockActionType is either lock or unlock, and a LockAction is a pair consisting of a LockActionType and a Lock on which to perform the given action. The function lock maps each object to its lock.

To see how the introduction of locks ensures that only one thread enters its synchronized region, we must consider the JLS rules for the concurrent behavior of locks.

JLS rules 5

1. [p. 403] "The actions performed by the main memory for any one lock are totally ordered; that is, for any two actions performed by the main memory on the same lock, one action precedes the other."
2. [p. 403] "Each lock or unlock action is performed jointly by some thread and the main memory."

Rule 5.1 supports our view of independent lock master agents. The actions on a single lock are ordered linearly, but actions on different locks may occur concurrently.

JLS rules 6 Let T be a thread and L be a lock.

1. [p. 406] "A lock action by T on L may occur only if, for every thread S other than T, the number of preceding unlock actions by S on L equals the number of preceding lock actions by S on L."
2. [p. 406] "An unlock action by thread T on lock L may occur only if the number of preceding unlock actions by T on L is strictly less than the number of preceding lock actions by T on L."

Rule 6.1 ensures that a thread's hold on a lock is exclusive, and Rule 6.2 ensures that for every unlock action there is a unique preceding lock action. But note that the rules allow several lock actions on the same lock to occur in succession, with no intervening unlock action. We find it convenient to think of a thread as building up a number of *claims* on a lock. A lock action adds a claim, and an unlock action takes one away. Rule 6.1 states that only one thread may

have a positive number of claims on the lock; furthermore, a thread may acquire multiple claims on a lock and surrenders the lock when and only when it has given up all claims. Rule 6.2 states that a thread may not release a claim on a lock it does not hold; the number of claims it has on a lock cannot be negative.

These rules ensure that the synchronization mechanism restricts synchronized code regions to one thread at a time. At most one thread at a time may have a positive number of claims on a given lock, and so if several threads need to lock the same lock, they must compete for it. Only the one with a positive number of claims on the lock is allowed into its synchronized region; the others must wait.

The function claims returns the number of claims a given thread has on a given lock. The function synchAction returns the action that the thread is requesting (if any).

A thread may only continue the execution of its program if it is active; *i.e.*, not synchronizing. We change the thread module accordingly, by guarding the rule *Execute program* with the term active?(Self).

term active?(t): undef?(synchAction(t))

To the rule *Execute program*, we add the options *Synch* and *Create object*. We add rules for threads' synchronization actions with locks. A thread synchronizes with a lock for either a lock action or an unlock action.

rule *Synch on ℓ to perform act*: synchAction(Self) := (act, ℓ)

rule *Synch*: **choose** *obj*: Object
choose *act*: LockActionType
Synch on lock(obj) to perform act

The rule *Create object* creates an object and associates it with a new lock. We also modify *Create thread* to associate each new thread with a new lock.

rule *Create object*: **extend** Object **with** *obj*
extend Lock **with** ℓ
lock(*obj*) := ℓ

The following JLS rules place some guarantees on the contents of a thread's working memory before and after synchronization actions.

JLS rules 7 [p. 407] "Let T be any thread, let V be any variable, and let L be any lock."

1. "Between an assign action by T on V and a subsequent unlock action by T on L, a store action by T on V must intervene; moreover, the write action corresponding to that store must precede the unlock action, as seen by main memory."
2. "Between a lock action by T on L and a subsequent use or store action by T on a variable V, an assign or load action on V must intervene; moreover, if it is a load action, then the read action corresponding to that load action must follow the lock action, as seen by main memory."

If a thread t issues an assign action A on a variable v and then issues an unlock action Ul on a lock ℓ, Rule 7.1 postulates that t issues a store action S on v (with corresponding write action W) such that $A < S < W < Ul$. Note that ℓ and v are independent; for an unlock action on a particular lock ℓ, Rule 7.1 applies to *all* variables v. Thus when a thread releases a claim on a lock, it is ensured that all its assigned values have been stored out and written to the main memory.

If a thread t issues a lock action Lk on a lock ℓ and then issues a use or store action US on a variable v, Rule 7.2 postulates that either t issues an assign action A on v such that $Lk < A < US$, or t issues a load action L (with corresponding read action R) on v such that $Lk < R < L < US$. For a lock action on a particular lock ℓ, Rule 7.2 applies to all variables v. So this rule ensures that by the time a thread acquires a claim on a lock, all the values cached in its working memory have been flushed out to main memory. When a thread acquires a claim on a lock, it acts as if its working memory is empty. Only values assigned or loaded in after the lock action may be used, and only values assigned after the lock action may be stored out.

The conditions imposed by these rules have some ramifications for earlier rules. A read message issued before a lock action cannot be loaded in after the lock action, but Rule 2.1 dictates that any such message must be loaded in. Therefore, all pending read messages must be loaded in before a lock action. Also, a value assigned before a lock action cannot be stored out after the lock action, but Rule 3.2 dictates that it must be stored out. Therefore, all assigned values must be stored out before a lock action.

The actions of a lock master consist of granting a claim to a lock and taking one away.

module Lock: **choose among**
  choose t: Thread: lockOK?(t)
    Lock for t
  choose t: Thread: unlockOK?(t)
    Unlock for t

rule *Lock for* t: claims(t, Self) := claims(t, Self) + 1
  synchAction(t) := undef
  do-forall v: Var: usableValue?(t, v)
    usableValue?(t, v) := false

rule *Unlock for* t: claims(t, Self) := claims(t, Self) - 1
  synchAction(t) := undef

The terms lockOK? and unlockOK? determine whether a given action is allowed. A lock action for a thread t on a lock ℓ is allowed only if t is synchronizing for a lock action on ℓ. Rule 6.1 allows such a lock action only if all threads other than t have no claims on ℓ. Rules 3.2 and 7.2 require that all previously assigned values have been stored out. Rules 2.1, 2.3 and 7.2 require that all access mes-

sages have been loaded in, and Rules 4.1 and 7.2 require that all access messages from volatile variables have been used.

term lockOK?(t):
synchAction(t) = (lock, Self) and (($\forall t'$: Thread: $t' \neq t$) claims(t', Self) = 0)
and (not $\exists v$: Var) freshAssign?(t, v) or ($\exists m$: AccMsg(v, t))
and (not $\exists v$: VolVar) usableValue?(t, v)

term unlockOK?(t):
synchAction(t) = (unlock, Self) and claims(t, Self) > 0
and (not $\exists v$: Var) freshAssign?(t, v) or ($\exists m$: UpdMsg(t, v))

5 Prescient store actions

A standard store action sends a value to the main memory from the working memory, where it was put by a preceding assign action. Under certain circumstances, Java allows an optimization in which a store action may *precede* its corresponding assign action, allowing the value to be transmitted to the main memory sooner than otherwise possible. This type of store action is called *prescient*, as it must be known ahead of time what the value of the assign action will be.

A prescient store action differs from a normal store action in that the value it sends to the main memory is not the current contents of working memory, but rather some fresh value. We call the following assign action *retroactive*, since the value it puts into working memory is not a fresh value, but rather the value of the preceding prescient store action. We find it natural to speak of prescient store actions and retroactive assign actions as distinct from regular store and assign actions.

We modify the ASM to model prescient store actions. If a given thread has issued a prescient store action on a given variable, without a corresponding retroactive assign action, the function presStoreVal returns the value of the prescient store.

We define rules for the new actions of prescient store and retroactive assign. Rule 8.4 disallows a store action on v. We also modify the rules *WM-EE transfer* and *WM-MM transfer*, adding prescient store and retroactive assign as options.

rule *Store presciently to* v: **choose** v: Var: presStoreOK?(Self, v)
freshAssign?(Self, v) := false
choose val: Value
presStoreVal(Self, v) := val
extend UpdMsg(Self, v) **with** m
val(m) := val

rule *Assign retroactively to* v: usableValue?(Self, v) := true
presStoreVal(Self, v) := undef
workingValue(Self, v) := presStoreVal(Self, v)

Prescient store actions must obey the following rules set out in the JLS:

JLS rules 8 [p. 408] "Suppose that a store by [a thread] T of [a non-volatile variable] V would follow a particular assign by T of V according to the rules of the previous sections, with no intervening load or assign by T of V. The special rule allows the store action to instead occur before the assign action, if the following restrictions are obeyed:

1. If the store action occurs, the assign is bound to occur.
2. No lock action intervenes between the relocated store and the assign.
3. No load of V intervenes between the relocated store and the assign.
4. No other store of V intervenes between the relocated store and the assign.
5. The store action sends to the main memory the value that the assign action will put into the working memory of thread T."

If a thread t performs a prescient store action PS on a variable v, there is a retroactive assign action $RA > PS$ by t on v such that RA assigns the value stored out at PS (Rules 8.1 and 8.5). Rules 8.1 and 8.5 ensure that the value t sends to main memory via a prescient store action ends up in t's working memory via a following retroactive assign action.

Furthermore, there is no action Act by t, where Act is a lock action or a load or store action on v, such that $PS < Act < RA$ (Rules 8.2-4). These rules ensure that relocating t's store action on v (*i.e.*, making it prescient) does not affect program behavior, in the sense that the effects of a prescient store action on t's working memory and the main memory are no different from those of the corresponding non-prescient store action.

The terms presStoreOK? and retroAssignOK? determine whether a prescient store action or retroactive assign action is allowed by the JLS rules, respectively. Rules 2.5 and 3.2 allow t to store to v presciently only if every AccMsg from v to t has been loaded in and there is no retroactive assign pending. Rule 8 restricts prescient store actions to volatile variables, and allows a retroactive assign action only if there is a preceding prescient store action without a corresponding retroactive assign action.

term presStoreOK?(v): not VolVar(v) and undef?(presStoreVal(Self, v))
and (not $\exists m$: AccMsg(v, Self))

term retroAssignOK?(v): def?(presStoreVal(Self, v))

Notice that t may issue a use action U on v between PS and RA. If the presciently stored value were to appear in t's working memory before t put it there at RA, t could end up using the presciently stored value prematurely. To prevent this, Rule 8.3 prohibits t from loading in from v between PS and RA. For the same reason, albeit less obviously, Rule 8.4 prohibits any lock action for t between PS and RA. This is needed because if a lock action for t were to occur and then t were to use v between PS and RA, Rule 7.2 would require a load action between PS and U, and such a load action is forbidden by Rule 8.3.

Relocating a store action makes sense only if there is no attempt to store between the relocated (prescient) store) action and its (retroactive) assign action. If t were to issue a store action S on v (with corresponding assign action A) between PS and RA, we would get the following undesirable result. S would follow PS, but the order of the corresponding assign actions would be different: RA would follow A. Thus the value stored through PS would be newer (*i.e.*, assigned more recently) than the value stored through the following action S. But Rule 2.5 would dictate that the newer value be overwritten in main memory by the older value. To prevent this, Rule 8.4 prohibits t from storing out a value of v between PS and RA.

Some of the rules introduced in previous sections (namely, 3.2–5 and 7.1–2) refer to store and assign actions and so must be modified to accommodate prescient store and retroactive assign actions. The rules as they appear in the JLS are not modified when prescient store actions are introduced.

We modify the terms lockOK? and loadOK? so that they enforce Rules 8.2–8.4. If a thread t has stored out presciently to v but has not yet performed the corresponding retroactive assign action, Rule 8.2 disallows any lock action for t, Rule 8.3 disallows a load action by t on v, and Rule 8.4 disallows any store action by t on v. To lockOK?(t), loadOK?(t) and storeOK?(t) we add the conjunct ($\forall v$: Var) undef?(presStoreVal(t, v).

Rule 8.1 ensures that if a prescient store action on a variable occurs, a corresponding retroactive assign action will occur sometime in the future. As with previous rules, this rule can be flouted by delaying a retroactive assign action indefinitely. We restrict attention to runs that avoid this situation.

6 `Thread` objects

We now consider how Java represents threads as objects. An object is an instance of a programmer-defined type called a *class*. Objects are created dynamically during a thread's computation. A class defines the state variables of its instances and the *methods* (operations) that may be invoked upon them. Each thread is represented by an object which is an instance of the class `Thread`. As there is a one-to-one correspondence between threads and their representative objects, we may identify a thread with its object. Information on a thread's state is held in its object. The methods defined in `Thread` allow a thread to access or modify its own state information and that of other threads.

We modify the ASM to model operations on threads. Since we identify a thread with its `Thread` instance, a member of the universe Thread is both a thread and a `Thread` instance. Note that `Thread` instances are members of both Thread and Object.

After a thread is created, it may be started by invoking the method `start` upon it. Once started, a thread is *alive* until it *stops*, which occurs when the thread's execution engine terminates or the method `stop` is invoked upon it. If `stop` is invoked on a thread, the thread *throws an exception*, signaling the occurrence of an unexpected event. Once a thread stops, it is no longer alive

and cannot be restarted. It is possible to stop a thread that has not started; if this happens, the thread will never become alive. It is not stated explicitly in the JLS what happens if `start` is invoked upon a thread that is already alive. However, the intent seems to be that the invoker of `start` throws an exception and nothing happens to the invokee [12].

We define functions started? and stopped? to represent the status of each thread. We also define rules *Start* and *Stop*, which update them appropriately.

The JLS rules impose some obligations on threads which they may not be able to fulfill after they have stopped. By Rules 2.1 and 2.3 (following the strict interpretation of "uniquely paired"), every access message must be loaded in. By Rule 4.1, every load action on a volatile variable must be followed by a corresponding use action, and by Rule 4.2, every assign action to a volatile variable must be followed by a corresponding store action. The JLS does not make it clear whether these obligations extend to threads that have stopped. Furthermore, it is not clear which choice is the more reasonable one. For instance, in some cases it may be desirable to have the last assign actions of a thread stored out, while in other cases it may be clearly undesirable; for example, consider a thread which is stopped because it is computing erroneous results. Officially, this is still an open issue [21]. For simplicity, we take the view that stopped threads are not obligated to do any further actions.

The JLS rules impose some obligations on variables which may not be fulfilled by the time the execution of all threads terminates; in particular, by Rule 2.4, every store message must be written. Thus it may still be necessary for variables to write some update messages in even after all threads have terminated.

A thread that is alive can be *suspended*, in which case it remains alive but its execution engine does nothing. A thread is suspended when some thread (possibly itself) invokes the method `suspend` upon it. A suspended thread *resumes* (becomes unsuspended) when some other thread invokes the method `resume` upon its `Thread` instance. It may be useful to suspend a thread when it cannot make progress by itself, freeing resources for other threads that can make progress. Can a suspended thread issue load or store actions? Officially, this is another unresolved issue [21]. Here we choose the more permissive interpretation, allowing suspended threads to issue read and load actions.

We add the function suspended? to represent the suspended status of each thread. We also define rules *Suspend* and *Resume*, which update this function.

A thread may be marked as a *daemon*. Such threads are intended to execute only in conjunction with non-daemon threads, performing "housekeeping" actions which assist non-daemon threads, but no actions useful in themselves. For this reason, once all non-daemon threads have stopped, execution of all threads halts. A program starts with a single non-daemon thread. A new thread starts with the daemon status of the thread that creates it, but its status can be changed via the `setDaemon` method.

To our ASM we add the function daemon?, which determines a thread's current daemon status, and the rule *Set daemon status*, which updates this function. We also modify the rule *Create thread*. A thread is added to the Object

universe and is initialized as unstarted, unstopped and unsuspended. It inherits the daemon status of the thread that created it.

We define a rule *Invoke thread method*, which chooses among the options *Start*, *Stop*, *Suspend* and *Set daemon status*. We modify the *Execute program* rule, adding the option *Invoke thread method*. To active? we add the conjuncts not suspended?(t) and undef?(synchAction(t)). We also modify the Thread module, guarding *Execute program* with alive?(Self) and not HaltAllThreads?

We define the term alive? to represent the running status of a thread. Also, the term HaltAllThreads? determines whether all non-daemon threads have terminated.

term alive?(t): started?(t) and not stopped?(t)

term HaltAllThreads?: ($\forall t$: Thread: alive?(t)) daemon?(t)

We modify the modules for variables and locks. If all non-daemon threads have terminated, the only action a variable may take is to write in update messages, so in the Var module we guard the read option with not HaltAllThreads?. There are no actions for locks to take once all non-daemon threads have terminated, so in the Lock module we guard the lock and unlock options with the term not HaltAllThreads?.

7 Waiting and notification

While a thread holds a particular lock, other threads in need of that lock are not allowed to proceed. In certain cases, a thread holding a lock may reach a state from which it cannot progress by itself; in such a case, suspending the thread may not be a solution, as the thread would still hold the lock even when suspended. It would be desirable for the thread to give up control and release its locks temporarily, so that other threads may acquire them.

Java has built-in support for this, through the mechanisms of *waiting* and *notification*. If a thread has a claim on the lock of an object, it may release all its claims on the lock and disable itself by invoking the method `wait` of class `Object` on the object. Another thread may signal to the waiting thread that further progress is possible through the `notify` or `notifyAll` methods. In this way, a waiting thread may resume execution when it is possible, without repeatedly checking its state. Each object has a *wait set*, empty when the object is created, which contains all threads waiting for the lock on the object.

The method `wait` is invoked by a thread t on an object obj. The thread t must hold the lock of obj; if it does not, an exception is thrown. Let k be the number of claims t has on the lock of obj; then k unlock actions are performed, and t is added to the wait set of obj and disabled. When t is re-enabled, it attempts to regain k claims on the lock; it may need to compete with other threads to do this. Once the lock claims are restored, the `wait` method terminates.

We modify the ASM to model waiting and notification actions. A thread goes through several phases while it waits on a lock. First it synchronizes with the

lock to release all claims on the lock. Then it waits to be notified by another thread. Once it is notified, it synchronizes with the lock to regain all the claims it released. For a thread that is waiting on a lock, the function waitMode gives the current phase of the thread, the function waitLock gives the lock on which it is waiting, and claimsToRegain gives the number of claims that it has (temporarily) released.

We introduce a rule for a thread's actions while waiting on a lock. Waiting involves synchronization with the lock during the unlock and relock phases. Note that the waiting thread does not change its own mode from wait to relock; this is done by another thread, through a notification.

rule *Start to wait*: **choose** *obj*: Object: claims(Self, lock(*obj*)) > 0
waitMode(Self) := unlock
waitLock(Self) := lock(*obj*)
claimsToRegain(Self) := claims(Self, lock(*obj*))

rule *Continue to wait*:
if waitMode(Self) = unlock **then**
  if claims(Self, waitLock(Self)) > 0 **then**
    Synch on waitLock(Self) to perform unlock
  else waitMode(Self) := wait
elseif waitMode(Self) = relock **then**
  if claims(Self, waitLock(Self)) < claimsToRegain(Self) **then**
    Synch on waitLock(Self) to perform lock
  else waitMode(Self) := undef

The method `notify` is also invoked by a thread t on an object obj. The thread t must hold the lock of obj; if it does not, an exception is thrown. If the wait set of obj is not empty, one thread is removed from it and enabled. The method `notifyAll` operates similarly but removes and enables all threads from the wait set. Neither method releases any of t's claims on the lock of obj, so there is no guarantee that the lock will be made available to the notified threads; this is left up to the programmer.

We introduce rules for notification. The method `notify` operates on a particular thread waiting on a particular lock; the method `notifyAll` operates on all threads waiting on a particular lock.

rule *Notify t*: waitMode(t) := relock

rule *Notify*:
choose *obj*: Object
  choose t: Thread: waitMode(t) = wait and waitLock(t) = lock(*obj*)
    Notify t

rule *NotifyAll*:
choose *obj*: Object
  do-forall t: Thread: waitMode(t) = wait and waitLock(t) = lock(*obj*)
    Notify t

When a thread stops, the method `notifyAll` is invoked upon its object. This allows a straightforward implementation of a technique called *thread joins*. Consider a scenario where a thread t' is expected to start executing only after another thread t has stopped. One way of implementing this is to have t' *join* with t by waiting until t has stopped before proceeding. Since `notifyAll` is invoked when a thread stops, t' may join with t by simply waiting on t's object. Since stopping a thread involves invoking `notifyAll` on its representative object, we change our ASM rule for stopping threads accordingly.

We point out a subtle issue. It is not clear from the JLS whether notification precedes stopping or *vice versa* when a thread stops. However, it seems reasonable that if stopping and notification do not occur as a single action, then notification must precede stopping (at least in the case where a thread stops itself). This leaves open the possibility that other threads may be notified of the thread's stop action before the thread has actually stopped, particularly if the notification process takes a relatively long time.

Threads may execute waiting or notification actions. Execution of a Java program does not proceed while a thread is waiting, so to the term active? we add the conjunct waitMode(t) $\neq$ wait. To the rule *Execute program*, we add the options *Start to wait*, *Notify* and *NotifyAll*. We also add the guard def?(waitMode(Self)); if this evaluates to true, then the rule *Continue to wait* fires; otherwise, one of the other options is chosen.

8 Conclusion

The Java concurrency model is currently in a state of flux. Researchers are proposing modifications to the model, to allow common compiler optimizations and programming practices that are currently prohibited by the model [19]. It is our belief that a satisfactory successor to the current model must start with a firm foundation. The specification of the model must not be subject to multiple interpretations, as is the description in the JLS. It is equally important that the specification be accessible to programmers who wish to use concurrency. While multithreaded programming is inherently complicated, the method of documentation should not exacerbate the problem.

We feel that this work has several things to offer the Java community. Our alternative view of Java concurrency brings to light some issues that might otherwise remain hidden in the JLS description. As the ASM model is mathematically precise, it can stand as an unambiguous cross-platform standard for the language. Our account of Java concurrency has an imperative flavor that is familiar to programmers, yet we are not restricted to a purely imperative approach; we are free to use a declarative style wherever it is more natural to do so. By implementing the concurrency model (albeit in a completely abstract way) we can readily see how the various constraints of the JLS definition interact with one another. This is much less obvious if constraints are presented in isolation, as they are in the JLS. Finally, programmers interested in multithreaded Java can explore the

model by using the ASMGofer prototype. We believe that ASMs are a useful specification tool for both the current and future models of Java concurrency.

References

1. I. Attali, D. Caromel, and M. Russo. A formal executable semantics for Java. In *Proceedings of the OOPSLA '98 Workshop on the Formal Underpinnings of Java*, 1998.
2. E. Börger. Why use Evolving Algebras for hardware and software engineering? In *Proceedings of SOFSEM*, 1995.
3. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer, 1998.
4. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From sequential to multi-threaded Java: An event-based operational semantics. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, pages 75–90. Springer, 1997.
5. E. Coscia and G. Reggio. A proposal for a semantics of a subset of multi-threaded "good" Java programs. In *Proceedings of the OOPSLA '98 Workshop on the Formal Underpinnings of Java*, 1998.
6. A. Gontmakher and A. Schuster. Java consistency: Non-operational characterizations for Java memory behavior. Technion/CS Technical Report CS0922, 1997.
7. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
8. Y. Gurevich. Evolving Algebras: an attempt to discover semantics. In G. Rozenberg and A. Salomå, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.
9. Y. Gurevich. Evolving Algebras 1993: Lipari guide. In E. Börger (editor), *Specification and Validation Methods*, Oxford University Press, 1995, 9–36.
10. Y. Gurevich. May 1997 draft of the ASM guide. Available at [22], 1997.
11. Y. Gurevich, W. Schulte and C. Wallace. Investigating Java concurrency using Abstract State Machines. Technical Report 2000-04, Department of Computer & Information Sciences, University of Delaware, 1999.
12. C. Horstmann and G. Cornell. *Core Java 1.1, volume II: Advanced Features*. Sun Microsystems Press, 1998.
13. Sun Microsystems. Java technology home page. `http://java.sun.com/`.
14. A. Jolin. Java's atomic assignment: The key to simpler data access across threads. *Java Report* 3(8), 27–36, 1998.
15. D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.
16. M. MacBeth, K. McGuigan and P. Hatcher. Executing Java threads in parallel in a distributed memory environment. In *Proceedings of IBM CASCON*, 1998.
17. S. Oaks and H. Wong. *Java Threads*. O'Reilly and Associates, 1997.
18. G. Plotkin. Structural Operational Semantics (Lecture notes). Technical Report DAIMI FN-19, Aarhus University, 1981.
19. W. Pugh. Fixing the Java memory model. In *Proceedings of ACM Java Grande*, 1999.
20. Joachim Schmid. ASMGofer home page. `http://www.tydo.de/AsmGofer/`.
21. G. Steele. Personal communication.
22. Univ. of Michigan. ASM home page. `http://www.eecs.umich.edu/gasm/`.
23. Univ. of Paderborn. ASM home page. `http://www.uni-paderborn.de/cs/asm/`.