New description of the Unified Memory Model Proposal for Java

Jeremy Manson, William Pugh and Sarita Adve

April 29, 2004, 9:35pm

0.1 Actions and Executions

An action a is described by a tuple $\langle t, k, v, u \rangle$, comprising:

t - the thread performing the action

k - the kind of action: volatile read, volatile write, (normal or non-volatile) read, (normal or non-volatile) write, lock or unlock. Volatile reads, volatile writes, locks and unlocks are synchronization actions. There are also external actions, and catatonia actions.

v - the variable or monitor involved in the action

u - an arbitrary unique identifier for the action

An execution E is described by a tuple $\langle P, A, \stackrel{po}{\rightarrow}, \stackrel{so}{\rightarrow}, W, V, \stackrel{sw}{\rightarrow}, \stackrel{hb}{\rightarrow}, \stackrel{ob}{\rightarrow} \rangle$, comprising:

P - a program

A - a set of actions

$\stackrel{po}{\rightarrow}$ - program order, which for each thread t, is a total order all actions performed by t in A

$\stackrel{so}{\rightarrow}$ - synchronization order, which is a total order over all synchronization actions in A

W - a write-seen function, which for each read r in A, gives $W(r)$, the write action seen by r in E.

V - a value-written function, which for each write w in A, gives $V(w)$, the value written by w in E.

$\stackrel{sw}{\rightarrow}$ - synchronizes-with, a partial order over synchronization actions.

$\stackrel{hb}{\rightarrow}$ - happens-before, a partial order over actions

$\stackrel{ob}{\rightarrow}$ - observable order, a total order over all actions that is consistent with the happens-before order and synchronization order.

Note that the synchronizes-with and happens-before are uniquely determined by the other components of an execution and the rules for well-formed executions.

Two of these kinds of actions need special descriptions.

external actions - An external action is an action that is observable outside of an execution. An external action tuple contains an additional component, which contains the results of the external action. This may be information as to the success or failure of the action, and any values read by the action.

Parameters to the external action (e.g., which bytes are written to which socket) are not part of the external action tuple. These parameters are set up by other actions within the thread and can be determined by examining the intra-thread semantics. They are not explicitly discussed in the memory model.

The primary impact of observable actions comes from the fact that if an external action is observed, it can inferred that other actions occur in a finite prefix of the observable order.

catatonia action - A catatonia action is only performed by a thread that is in an infinite loop in which no memory or observable actions are performed. If a thread performs a catatonia action, it will be followed by an infinite number of catatonia actions. These actions are introduced so that we can explain why such a thread may cause all other threads to stall and fail to make progress.

0.2 Happens-Before Edges

If we have two actions x and y, we use $x \stackrel{hb}{\rightarrow} y$ to mean that x *happens before* y. If x and y are actions of the same thread and x comes before y in program order, then $x \stackrel{hb}{\rightarrow} y$.

Synchronization actions also induce happens-before edges. We call the resulting edges *synchronized-with* edges, and they are defined as follows:

- There is a happens-before edge from an unlock action on monitor m to all subsequent lock actions on m (where subsequent is defined according to the synchronization order).
- There is a happens-before edge from a write to a volatile variable v to all subsequent reads of v by any thread (where subsequent is defined according to the synchronization order).
- There is a happens-before edge from an action that starts a thread to the first action in the thread it starts.
- There is a happens-before edge between the final action in a thread T1 and an action in another thread T2 that detects that T1 has terminated. T2 may accomplish this by calling `T1.isAlive()` or doing a join action on T1.
- If thread T1 interrupts thread T2, there is a happens-before edge from the interrupt by T1 to the point where any other thread (including T2) determines that T2 has been interrupted (by having an InterruptedException thrown or by invoking Thread.interrupted or Thread.isInterrupted).

In addition, we have three other rules for generating happens-before edges.

- There is a happens-before edge from the write of the default value (zero, false or null) of each variable to the first action in every thread. No actions happen-before a write to a variable of a default value.
- There is a happens-before edge from the end of a constructor of an object to the start of a finalizer for that object.
- Happens-before is transitively closed. In other words, if $x \stackrel{hb}{\rightarrow} y$ and $y \stackrel{hb}{\rightarrow} z$, then $x \stackrel{hb}{\rightarrow} z$.

0.3 Definitions

1. **Definition of synchronizes-with** If $x \stackrel{so}{\rightarrow} y$ and x is a volatile write or an unlock, and y is a volatile read of the same variable as x, or a lock of the same monitor as x, then $x \stackrel{sw}{\rightarrow} y$. Volatile writes and unlocks are referred to as *releases*, and volatile reads and locks are referred to as *acquires*.
2. **Definition of happens-before** The happens-before order $\stackrel{hb}{\rightarrow}$ is the transitive closure of $\stackrel{sw}{\rightarrow} \cup \stackrel{po}{\rightarrow}$.
3. **Definition of sufficient synchronization edges**. A set of synchronization edges is *sufficient* if it is the minimal set such that you can take the transitive closure of those edges with program order edges, and determine all of the happens-before edges in the execution. This set is unique.
4. **Restrictions of partial orders and functions** We use $f|_d$ to denote the function given by restricting the domain of f to d: for all $x \in d$, $f(x) = f|_d(x)$ and for all $x \notin d, f(x) = \perp$. Similarly, we use $\stackrel{e}{\rightarrow}|_d$ to represent the restriction of the partial order $\stackrel{e}{\rightarrow}$ to the elements in d: for all $x, y \in d$, $x \stackrel{e}{\rightarrow} y$ if and only if $x \stackrel{e}{\rightarrow}|_d y$. If either $x \notin d$ or $y \notin d$, then it is not the case that $x \stackrel{e}{\rightarrow}|_d y$.

0.4 Well-formed Executions

We only consider well-formed executions. An execution $E = \langle P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow} \overset{ob}{\rightarrow} \rangle$ is well formed if the following conditions are true:

1. **Each read sees a write in the execution. All volatile reads see volatile writes, and all non-volatile reads see non-volatile writes.** For all reads $r \in A$, we have $W(r) \in A$ and $W(r).v = r.v$. If $r.k$ is a volatile read, then $W(r).k$ is a volatile write, otherwise $r.k$ is a normal read, and $W(r).k$ is a normal write.

2. **Happens-before order is acyclic**. The transitive closure of synchronization order and program order is acyclic.

3. **The execution obeys intra-thread consistency.** For each thread t, the actions performed by t in A are the same as would be generated by that thread in program-order in isolation, with each write w writing the value $V(w)$, given that each read r sees the value $V(W(r))$. values seen by each read are determined by the memory Note that intrathread semantics predict by intrathread semantics, but take for granted the values seen by reads. The program-order must reflect the program order in which the actions would be performed according to the intrathread semantics of P.

4. **The execution obeys happens-before consistency.** For all reads $r \in A$, it is not the case that $r \overset{hb}{\rightarrow} W(r)$ or that there exists a write $w \in A$ such that $w.v = r.v$ and $W(r) \overset{hb}{\rightarrow} w \overset{hb}{\rightarrow} r$.

5. **The execution obeys synchronization-order consistency.** For all volatile reads $r \in A$, it is not the case that $r \overset{so}{\rightarrow} W(r)$ or that there exists a write $w \in A$ such that $w.v = r.v$ and $W(r) \overset{so}{\rightarrow} w \overset{so}{\rightarrow} r$.

0.5 Observable External Actions

An execution may have an uncountably infinite number of actions. This models a non-terminating execution. In an infinite execution, the only external actions that can be observed are those that occur in a finite prefix of the observable order.

Since we can have an uncountable infinity of actions, that means an execution may contain an (unobservable) action x such that an infinite number of actions occur before x in the observable order.

0.6 Executions valid according to the Java Memory Model

A well-formed execution $E = \langle P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow}, \overset{ob}{\rightarrow} \rangle$ is validated by committing actions from A. If all of the actions in A can be committed, then the execution is valid according to the Java memory model.

Starting with the empty set as C_0, we perform several steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E_i containing C_i that meets certain conditions.

Formally, there exists

- Sets of actions $C_0, C_1, \ldots, C_n$ such that
  - $C_0 = \emptyset$
  - $C_i \subseteq C_{i+1}$
  - $C_n = A$
- Well-formed executions $E_1, \ldots, E_n$, where $E_i = \langle P, A_i, \overset{po_i}{\rightarrow}, \overset{so_i}{\rightarrow}, W_i, V_i, \overset{sw_i}{\rightarrow}, \overset{hb_i}{\rightarrow}, \overset{ob_i}{\rightarrow} \rangle$.

Given these sets of actions $C_0 ... C_n$ and executions $E_1 ... E_n$, every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally,

1. $C_i \subseteq A_i$
2. $\overset{hb_i}{\rightarrow}|_{C_i} = \overset{hb}{\rightarrow}|_{C_i}$
3. $\overset{so_i}{\rightarrow}|_{C_i} = \overset{so}{\rightarrow}|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. Only the reads in C_{i-1} need to see the same writes in E_i as in E. Formally,

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. Each read r in $C_i - C_{i-1}$ must see writes in C_{i-1} in both E_i and E, but may see a different write in E_i from the one it sees in E. Formally,

6. For any read $r \in A_i - C_{i-1}$, we have $W_i(r) \overset{hb_i}{\rightarrow} r$
7. For any read $r \in C_i - C_{i-1}$, we have $W_i(r) \in C_{i-1}$ and $W(r) \in C_{i-1}$

Given a set of sufficient synchronizes-with edges for E_i, if there is a release-acquire pair that happens-before an action you are committing, then that pair must be present in all E_j, where $j \geq i$. Formally,

8. Let $\overset{ssw_i}{\rightarrow}$ be the $\overset{sw_i}{\rightarrow}$ edges that are also in the transitive reduction of $\overset{hb_i}{\rightarrow}$ but not in $\overset{po_i}{\rightarrow}$. We call $\overset{ssw_i}{\rightarrow}$ the sufficient synchronizes-with edges for E_i. If $x \overset{ssw_i}{\rightarrow} y \overset{hb_i}{\rightarrow} z$ and $z \in C_i$, then $x \overset{sw_j}{\rightarrow} y$ for all $j \geq i$.
9. If x is an external action, $x \overset{hb_i}{\rightarrow} y$ and $y \in C_i$, then $x \in C_i$.

0.7 Finalization

Each execution has a number of *reachability decision points*, labeled d_i. Each action either comes before d_i or comes after d_i. Other than as explicitly mentioned, *comes before* in this section is unrelated to all other orderings in this document.

If r is a read that sees a write w and r comes before d_i, then w must come before d_i. If a release-acquire pair $\langle r, a \rangle$ is contained in the set of sufficient synchronization edges for an execution and a comes before d_i, then r must come before d_i.

At each reachability decision point, some set of objects are marked as unreachable, and some subset of those objects are marked as finalizable.

Reachability

An object B is definitely reachable from A at d_i if and only if

- there is a write $w1$ to an element v of A that is a reference to B and there does not exist a write $w2$ to v s.t. $\neg(w2 \overset{hb}{\rightarrow} w1)$, and both $w1$ and $w2$ come before d_i, or
- there is an object C such that C is definitely reachable from A and B is definitely reachable from C

A use of X is active if

- it reads or writes an element of X
- it synchronizes on X
- it writes a reference to X
- it is an active use of an object Y, and X is definitely reachable from Y
- X is definitely reachable from a static field of a class loaded by a definitely reachable classloader

If an object X is marked as unreachable at d_i,

- X must not be definitely reachable at d_i,
- All active uses of X in thread t that come after d_i must occur as a result of thread t performing a read that comes after d_i of a reference to X or in the finalizer invocation for X.
- All reads that come after d_i that see a reference to X must see writes to elements of objects that were unreachable at d_i, or see writes that came after d_i.

If an object X marked as finalizable at d_i, then

- X must be marked as unreachable at d_i,
- d_i must be the only place where X is marked as finalizable,
- actions that happen-after the finalizer invocation must come after d_i