New description of the Unified Memory Model Proposal for Java

Jeremy Manson, William Pugh and Sarita Adve

April 5, 2004, 9:25pm

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.

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 $\langle P, A, \stackrel{po}{\rightarrow}, \stackrel{so}{\rightarrow}, W, V, \stackrel{sw}{\rightarrow}, \stackrel{hb}{\rightarrow} \rangle$, 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

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.

Definitions

1. **Definition of synchronizes-with** If $x \stackrel{so}{\to} y$ 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 $x \stackrel{sw}{\to} y$. 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 $\stackrel{hb}{\to}$ is the transitive closure of $\stackrel{sw}{\to} \cup \stackrel{po}{\to}$.

3. **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 $x \notin d, f(x) = \bot$. Similarly, we use $\stackrel{e}{\to}|_d$ to represent the restriction of the partial order $\stackrel{e}{\to}$ to the elements in d: for all $x, y \in d$, $x \stackrel{e}{\to} y$ if and only if $x \stackrel{e}{\to}|_d\, y$. If either $x \notin d$ or $y \notin d$, then it is not the case that $x \stackrel{e}{\to}|_d\, y$.

Well-formed Executions

We only consider well-formed executions. An execution $E = \langle P, A, \stackrel{po}{\to}, \stackrel{so}{\to}, W, V, \stackrel{sw}{\to}, \stackrel{hb}{\to}\rangle$ 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. **Synchronization order is consistent with program order** There do not exist $x, y \in A$, such that $x \stackrel{so}{\to} y \wedge y \stackrel{po}{\to} x$. 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 $V(w)$ and each read r seeing the value $V(W(r))$. The program-order must reflect the program order of P.

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

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

Executions valid according to the Java Memory Model

A well-formed execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$ 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 $E_i = \langle P, A_i, \xrightarrow{po_i}, \xrightarrow{so_i}, W_i, V_i, \xrightarrow{sw_i}, \xrightarrow{hb_i} \rangle$.

Given these sets of actions $C_0 \ldots C_n$ and executions $E_1 \ldots E_n$, 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. $\xrightarrow{hb_i}|_{C_i} = \xrightarrow{hb}|_{C_i}$
3. $\xrightarrow{so_i}|_{C_i} = \xrightarrow{so}|_{C_i}$

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. All reads in $C_i - C_{i-1}$ must see writes in C_{i-1} in both E_i and E. Formally,

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

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 program. This set is unique.

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 $\xrightarrow{ssw_i}$ be the $\xrightarrow{sw_i}$ edges that are also in the transitive reduction of $\xrightarrow{hb_i}$. We call $\xrightarrow{ssw_i}$ the sufficient sychronizes-with edges for E_i. If $x \xrightarrow{ssw_i} y \xrightarrow{hb_i} z$ and $z \in C_i$, then $x \xrightarrow{sw_j} y$ for all $j \geq i$.