# Formal Description of the Manson/Pugh Model

February 6, 2004, 1:44pm

This document is intended to supplement the Public Review of the Java Memory Model. It can be considered as a replacement for Sections 7 and 8 of that document. It is strongly recommended that readers familiarize themselves with that material before reading this. In particular, the definitions in Section 5 and the discussion of memory model issues in Section 7 are frequently referred to here.

## 1 Causality

Happens-Before Consistency is a necessary, but not sufficient, set of constraints. In other words, we need the requirements imposed by Happens-Before Consistency, but they allow for unacceptable behaviors.

In particular, one of our key requirements is that correctly synchronized programs may exhibit only sequentially consistent behavior. Happens-Before Consistency alone will violate this requirement. Remember that a program is correctly synchronized if, when it is executed in a sequentially consistent manner, there are no data races among its non-volatile variables.

Consider the code in Figure 1. If this code is executed in a sequentially consistent way, each action will occur in program order, and neither of the writes will occur. Since no writes occur, there can be no data races: the program is correctly synchronized. We therefore only want the program to exhibit sequentially consistent behavior.

Could we get a non-sequentially consistent behavior from this program? Consider what would happen if both $r1$ and $r2$ saw the value 1. Can we argue that this relatively nonsensical result is legal under Happens-Before Consistency?

The answer to this is "yes". The read in Thread 2 is allowed to see the write in Thread 1, because there is no happens-before relationship to prevent it. Similarly, the read in Thread 1 is allowed to see the read in Thread 2: there is no synchronization to prevent that, either. Happens-Before Consistency is therefore inadequate for our purposes.

Even for incorrectly synchronized programs, Happens-Before Consistency is too weak: it can allow situations in which an action causes itself to happen, when it could happen no other way. We say that an execution that behaves this way contains a *causal loop*. At the extreme, this might allow a value to appear out of thin air. An example of this is given in Figure 2. If we decide arbitrarily that the writes in each thread will be of the value 42, the behavior `r1 == r2 == 42` can be validated as hb-consistent: each read sees a write in the execution, without any intervening happens-before relation.

Initially, x == y == 0

| Thread 1 | Thread 2 |
|---|---|
| r1 = x; | r2 = y; |
| if (r1 != 0) | if (r2 != 0) |
| y = 1; | x = 1; |

Correctly synchronized, so r1 == r2 == 0 is the only legal behavior

Figure 1: Happens-Before Consistency is not sufficient

Initially, x == y == 0

| Thread 1 | Thread 2 |
|---|---|
| r1 = x; | r2 = y; |
| y = r1; | x = r2; |

Incorrectly Synchronized: But r1 == r2 == 42 Still Cannot Happen

Figure 2: Motivation for disallowing some cycles

To avoid problems such as this, we require that executions respect causality. It turns out that formally defining causality in a multithreaded context is tricky and subtle.

## 1.1 Justification Orders

For any execution trace, we require the existence of a *justification order*, which is an ordered list of the reads, writes and synchronization actions in that execution. The justification order can be considered an explanation of how an execution happened. For example, first we can show that $x$ could happen; once we know $x$ will happen, we can show that $y$ can happen; once we know that both $x$ and $y$ will happen, we can show that $z$ can happen. In other words, it is a linear sequence of causes and effects.

The justification order does *not* have to be consistent with the program order or the happens-before order. Any total order over actions in an execution trace is potentially a valid justification order. The justification order could, for example, reflect the order in which the actions might occur after compiler transformations have taken place.

The intuition behind justification orders is that for each prefix of that justification order, the next action in the order is *justified* by the actions in the prefix. Actions that do not involve potential causal loops do not need to be explicitly justified.

How do we determine which actions need to be justified? In every causal loop, there must be some action that appears to occur before some action that happens before it. In the case of Figure 2, for example, the write to $x$ appears to occur before the read of $y$, even though the read happens before the write in program order. We refer to such actions as *prescient*; it is these that we have to justify.

Formally, an action $x$ in a trace $\langle S, so, \stackrel{hb}{\rightarrow}, jo \rangle$ is prescient if and only if there exists an action $y$ that occurs after $x$ in the justification order such that $y \stackrel{hb}{\rightarrow} x$.

All prescient actions must be justified. To justify a prescient action $x$ in trace $E$, let $\alpha$ be the actions that occur before $x$ in the justification order. We need to show that $x$ is allowed to occur in all executions with a justification order that starts with $\alpha$ and contains no prescient actions after $\alpha$. Section 1.4 describes an alternative weak causality model, in which we would only require that there exist some execution in which $x$ would be allowed to occur.

It should be fairly clear that there are no justification orders for which Figure 2 will produce 42: there is no sequence of actions that will guarantee that 42 will be written to $x$ or $y$.

In addition, the reads in Figure 1 will not be able to see the value 1. The first action in the justification order would have to be a write of 1 or a read of 1. Since neither of those are allowed in any non-prescient execution, they cannot be the first action in a justification order.

Formally defining causality is somewhat involved. To do so, it is necessary to define what it means for one action to be in more than one execution. This definition will be omitted here; the full definition can be found, along with the formal model, in Appendix 2.

## 1.2 When are Actions Justified to Occur in the Justification Order?

### 1.2.1 Reads

We need to state more formally what it means for a read $r$ to be justified in occurring in an execution $E$ whose justification order is $\alpha r \beta$. If $r$ is involved in a data race, then execution can non-deterministically choose which of the multiple writes visible to $r$ is seen by $r$, which makes it difficult to guarantee that $r$ will see a particular value.

Because we cannot guarantee that a particular value is seen, we make a weaker guarantee about when a read is justified. The read $r$ is justified if, in all executions whose justification order consists of $\alpha$ followed by non-prescient actions, there is a corresponding read $r'$ that is allowed to observe the same value that $r$ observed.

Intuitively, we are only able to justify a read if it is always *allowed to occur* based on the actions we have already justified.

Once we have demonstrated that a read can occur, we need to establish what values that read can see. To guarantee causality, we require that a read only see writes that occurred earlier in the justification order.

Finally, it should be noted that reads of volatile variables are only allowed to see the last write to that variable in the synchronization order.

### 1.2.2 Writes

We need to explore the consequences of allowing writes to be performed presciently more fully. Consider a program that is correctly synchronized, in which a particular write always happens when the program is executed non-presciently. That write can now be performed presciently (let's call the resulting execution $E'$). What happens if performing the write presciently allows some non-sequentially consistent behavior?

Before compiler transformation

Initially, a = 0, b = 1

| Thread 1 | Thread 2 |
|---|---|
| 1: r1 = a; | 5: r3 = b; |
| 2: r2 = a; | 6: a = r3; |
| 3: if (r1 == r2) | |
| 4: b = 2; | |

After compiler transformation

Initially, a = 0, b = 1

| Thread 1 | Thread 2 |
|---|---|
| 4: b = 2; | 5: r3 = b; |
| 1: r1 = a; | 6: a = r3; |
| 2: j = r1; | |
| 3: if (true) ; | |

Is r1 == r2 == r3 == 2 possible? r1 == r2 == r3 == 2 is sequentially consistent

Figure 3: Motivation for allowing forbidden reads

In such a case, there is a read that happened before the write in all the non-prescient executions that does not happen before the write in $E'$. To prevent this, we make a simple rule: all such reads must also happen before the write in $E'$.

## 1.3 Forbidden Sets

In order to perform an action presciently, we must be guaranteed that the action will occur. In most programs, there are many actions that do not have this guarantee; given the full freedom of the JMM, some actions will not always occur. However, compiler transformations may modify the program so that those actions are guaranteed to happen. After such a compiler transformation, we should be able to perform such actions presciently.

In Figure 3, we see an example of such a transformation. The compiler can

- eliminate the redundant read of a, replacing `2` with $r2 = r1$, then
- determine that the expression $r1 == r2$ is always true, eliminating the conditional branch `3`, and finally
- move the write `4: b = 2` early.

Here, the assignment `4: b = 2` is always guaranteed to happen, because the reads of $a$ always return the same value. Without this information, the assignment seems to cause itself to happen. Thus, simple compiler optimizations can lead to an apparent causal loop without a workable justification order. We must allow these cases, but also prevent cases where, if $r1 \neq r2$, $r3$ is assigned a value other than 1.

To validate such an execution we would need a justification order that makes $r1 == r2 == r3 == 2$ a causally consistent execution of Figure 3. To see this behavior, we need a justification order over valid executions that would justify this behavior in an execution.

Under the model as it stands, how would we go about trying to construct a justification order to validate this behavior? In this case, we are trying to capture a potential behavior of the transformed program: the case where `4` happens first, then all of Thread 2, and finally `1 - 3`. This would suggest { `4, 5, 6, 1, 2, 3` } as a potential justification order.

However, we cannot use this justification order assuming only Causality and Happens-Before Consistency. The prefix of `4 (b = 2)` is empty, so all of the validated executions for which the empty set is a prefix (i.e., all validated executions) must allow the write `4` to occur. The problem is that `4` is not guaranteed to occur in all non-prescient executions; it only occurs when $r1$ and $r2$ see the same value. If we were able to exclude all executions in which $r1$ and $r2$ see different values, then we could use the justification order { `4, 5, 6, 1, 2, 3` }

In short, compiler transformations can make certain executions (such as the ones in which `1` and `2` do not see the same value) impossible. This prohibition, in turn, can lead to additional executions that seem cyclic.

For the purposes of showing that a prescient action $x$ is justified, a set of behaviors that are not possible on a particular implementation of a JVM may be specified. This, in turn, allows other actions to be guaranteed and performed presciently, allowing for new behaviors.

However, this behavior must be tempered. If we allowed arbitrary executions to be forbidden, we could conceivably, for example, forbid all executions. We could then vacuously justify any action we liked, because it would occur in every execution. This sort of behavior is nonsensical; we therefore cannot allow arbitrary executions to be forbidden.

Instead of allowing arbitrary executions to be forbidden, we forbid a set $F$ of justification order prefixes. If an execution's justification order begins with an element of this set, it is forbidden.

We further require that each forbidden execution have a legal *alternate execution*. We describe prefixes $\alpha x$, where $\alpha$ is a sequence of actions, and $x$ is the last element in the prefix. For each $\alpha x \in F$ there exists some alternate, non-forbidden execution $E$ with a justification order $\alpha\beta$ such that $\beta$ contains no prescient actions.

Finally, we wish to say that if a particular execution is not forbidden, then executions that are identical to it cannot be forbidden. We construct identical executions by performing a *prescient relaxation* of an execution $E = \alpha xy\beta$. If

- $x$ and $y$ are not both synchronization actions,
- $x$ is prescient, $y$ is not, and
- $x$ is not a write seen by $y$.

then the prescient relaxation of $x$ in $E$ gives an execution $E'$ that is identical to $E$, except that the justification order of $E'$ is $\alpha yx\beta$. An execution $E$ is forbidden if any prescient relaxation of $E$ starts with a forbidden prefix.

Using forbidden executions, we can show that the execution in Figure 3 respects causality. This can be done by forbidding all executions where `h1` and `h2` do not return the same value. Execution traces where they do return the same value can be provided as alternate executions.

## 1.4 Causality Model Summary

Figure 4 summarizes, using the informal notation used in this section, the causality model proposed for the Java Memory Model. A more formal treatment can be found in Appendix 2.

For every execution, there is a total order over actions, consistent with the synchronization order, called the *justification order*.
Any read action must see a write that occurs earlier in the justification order. A volatile read always sees the result of the last volatile write in the justification order.
An action $x$ is *prescient* if there exists an action $y$ that occurs after $x$ in the justification order such that $y \stackrel{hb}{\rightarrow} x$. Each prescient action $x$ in an execution $E$ must be justified. Let $\alpha$ be the sequence of actions that precedes $x$ in the justification order of $E$. Let $J$ be the set of all non-forbidden hb-consistent executions whose justification order consists of $\alpha$ followed by non-prescient actions. To prove $x$ is justified, we need to show that for each $E'$ in $J$ it must have an action $x'$ such that:

- $x'$ is congruent to $x$; specifically, either $x'$ and $x$ are the same action, or they are both reads of the same variable and it would be hb-consistent for $x'$ to see the write seen by $x$, and
- (Prescient Write Rule) if $x$ is a write, then for each thread $t$, let $c$ be the number of reads in $E'$ performed by $t$ that conflict with $x'$ and happen-before $x'$. At least $c$ reads that conflict with $x$ and happen-before $x$ must be performed by $t$ in $E$.

Justification may involve the use of forbidden executions. See Figure 5 for details.
Given these definitions, an hb-consistent execution $E$ is legal if and only if there exists a set of forbidden prefixes $F$ such that $E$ is not forbidden by $F$ and using $F$ as the forbidden prefixes, all of the prescient actions in $E$ are justified.

Figure 4: Strong Causality Model for the Java Memory Model

The *prescient relaxation* of $x$ in $E$, where $E = \alpha xy\beta$, gives an execution $E'$ that is identical to $E$, except that the justification order of $E'$ is $\alpha yx\beta$. To perform a prescient relaxation of $\alpha xy\beta$, it is necessary that

- $x$ and $y$ not be both synchronization actions,
- $x$ be prescient and $y$ not be prescient, and
- $x$ not be a write seen by $y$.

Forbidden executions are defined by a set of forbidden justification order prefixes $F$. For each forbidden prefix $\alpha x$, the action $x$ must be non-prescient and either be a read or a synchronization action. Given $F$, an execution $E$ is forbidden by $F$ if any application of zero or more applications of prescient relaxation to $E$ generates an execution trace whose justification order starts with a forbidden prefix (typically, $F$ is empty and no executions are forbidden).
A set of forbidden prefixes must be valid. To show that a set of forbidden prefixes is valid, we must show that for each prefix $\alpha x \in F$, we have the following constraints:

- If $x$ is a read, either:
  - There exists some non-forbidden execution $E$ with a justification order $\alpha x'\beta$ such that $\beta$ contains no prescient actions, and $x'$ is a read corresponding to $x$ (a read by the same thread of the same variable, but a different write in $\alpha$ of a different value), or
  - Without considering $\alpha x$ as a forbidden prefix, there exists a non-forbidden execution $E$ with a justification order $\alpha' w' x' \beta'$ such that $\beta$ contains no prescient actions and $x'$ sees $w'$.
- If $x$ is a synchronization action, there exists some non-forbidden execution $E$ with a justification order $\alpha x'\beta$ such that $\beta$ contains no prescient actions, and $x'$ must be a different synchronization action (by another thread).

Figure 5: Forbidden Execution Definition

# 2 Formal Model of Strong Causality

This section describes, in more detail, the formal model for Strong Causality of Java programs. It includes formal definitions for what it means for an action to be present in multiple executions. In addition, the full semantics for strong causality of a program $P$ are detailed in Figure 6.

## 2.1 Definitions

To use justification orders, we must first define what it means for two actions to correspond to each other in two different executions.

**Congruence** First, we define a property called *congruence*. Two justification orders are congruent to each other if

- The justification orders are the same length, and
- All of the elements in each of the justification orders are the same, and
- If the $i^{th}$ element of the justification order that justifies one happens before the action, then the $i^{th}$ element of the justification order that justifies the other happens before the action.

For two justification orders $\alpha$ and $\alpha'$, this is written $\alpha \cong \alpha'$.

**Equivalence** We also define what it means for two executions to be *equivalent*. The justification order $\alpha'$ of an execution is equivalent to a justification order $\alpha$ (written $\alpha \equiv \alpha'$) if $\alpha \cong \alpha'$ and all of the information with which $\alpha_i$ is annotated (including the monitor accessed, the variable read or written and the value read or written) is the same as that for $\alpha'_i$.

**Correspondence** Finally, we can say what it means for one action to be in two different executions, given prefixes $\alpha$ and $\alpha'$ for those actions. We say that $\alpha x \mapsto \alpha' x'$ (read $\alpha x$ *corresponds* to $\alpha' x'$ if:

- $\alpha \equiv \alpha'$
- $\alpha x \cong \alpha' x'$
- If $x$ is a read, $x'$ is allowed to read the same value,
- If $x'$ is a write, it writes the same value as $x$, and
- $x$ and $x'$ act on the same variable.

The formal statement of these definitions can be found in Figure 7.

$$
\begin{array}{l}
E = \langle S, so, \xrightarrow{hb}, jo\rangle \in \text{valid} \iff E \in \text{hb-consistent} \wedge \\
\quad \exists F : F \text{ is legal} \wedge E \notin \text{forbidden}(F) \wedge \\
\quad\quad \forall x \in jo, jo = \alpha x \beta \wedge \\
\quad\quad\quad x \text{ is a read of write } \ w \Rightarrow w \in \alpha \wedge \\
\quad\quad\quad x \in \text{prescient}_E \Rightarrow \\
\quad\quad\quad\quad \textbf{let } J = \{\langle S', so', \xrightarrow{hb'}, \alpha'\beta'\rangle \mid \\
\quad\quad\quad\quad\quad\quad \langle S', so', \xrightarrow{hb'}\rangle \in \text{hb-consistent} \\
\quad\quad\quad\quad\quad\quad \wedge\ \alpha \equiv \alpha' \\
\quad\quad\quad\quad\quad\quad \wedge\ \beta' \text{ does not contain prescient actions } \} \\
\quad\quad\quad\quad\ \textbf{in } \ \forall E' = \langle S', so', \xrightarrow{hb'}, \alpha'\beta'\rangle \in J - \text{forbidden}(F) \\
\quad\quad\quad\quad\quad\quad \exists x' \in \beta' : \alpha x \mapsto \alpha' x' \\
\quad\quad\quad\quad\quad\quad \wedge\ x \text{ is a write of } \ v \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad \forall \text{ threads } t \in E, \text{ threads } t' \in E' : \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \{r \mid r \in t' \wedge r \xrightarrow{hb'} \ x'\} \mid \leq \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \{r \mid r \in t \wedge r \xrightarrow{hb} \ x\} \mid
\end{array}
$$

A set of forbidden prefixes $F$ is a set of prefixes of justification orders.
A set of forbidden prefixes $F$ is legal if and only if for each hb-consistent execution $E = \langle S, so, \xrightarrow{hb}, \alpha x \beta\rangle$ such that $\alpha x \in F$, either its justification order is infinite, or

$$
\begin{array}{l}
\exists E' = \langle S', so', \xrightarrow{hb'}, \alpha' x' \beta'\rangle \in \text{hb-consistent} \wedge \\
\quad \alpha \equiv \alpha' \wedge \\
\quad x'\beta' \text{ does not contain prescient actions } \wedge \\
\quad \alpha' x' \notin F \wedge \\
\quad x \text{ is a read } \Rightarrow \alpha x \mapsto \alpha' x' \wedge \\
\quad x \text{ is a synchronization action } \Rightarrow \\
\quad\quad x' \text{ is a synchronization action}
\end{array}
$$

Given $F$, we define forbidden($F$) to be the set of executions forbidden by $F$:

$$\text{forbidden}(F) = \{E' \mid E = \langle S, so, \xrightarrow{hb}, \alpha x \beta\rangle \wedge \alpha x \in F \wedge E' \in \text{pr}^*(E)\}$$

Figure 6: Full Semantics

Given $E = \langle S, so, \stackrel{hb}{\rightarrow}, \alpha x \beta \rangle$, $E' = \langle S', so', \stackrel{hb'}{\rightarrow}, \alpha' \beta' x' \gamma' \rangle$,

- $\alpha \cong \alpha' \iff$
  - $\text{length}(\alpha) = \text{length}(\alpha')$
  - $\forall i, j, 0 \leq i, j < \text{length}(\alpha) : \alpha_i \stackrel{hb}{\rightarrow} \alpha_j \iff \alpha'_i \stackrel{hb'}{\rightarrow} \alpha'_j$
- $x = y \iff$ all of the information with which $x$ is annotated (including the thread performed by, the monitor accessed, the variable read or written and the value read or written) is the same as that for $y$.
- $\alpha \equiv \alpha' \iff$
  - $\alpha \cong \alpha'$
  - $\forall i, 0 \leq i < \text{length}(\alpha) : \alpha_i = \alpha'_i$
- $\alpha x \mapsto \alpha' x' \iff$
  - $\alpha \equiv \alpha'$
  - $\alpha x \cong \alpha' x'$
  - $x = x'$, except that if $x$ is a read, then $x'$ doesn't seen to see the same write as $x$, but only be able to observe the same write as $x$
- $x \in \text{prescient}_E \iff$
  - $\exists y \in \beta : y \stackrel{hb}{\rightarrow} x$

Given $E = \langle S, so, \stackrel{hb}{\rightarrow}, \alpha x y \beta \rangle$, $E' = \langle S', so', \stackrel{hb}{\rightarrow}\ ', \alpha' y' x' \beta' \rangle$

$E' \in \text{pr}(E) \iff$
$\alpha x y \beta \equiv \alpha' x' y' \beta' \wedge$
$x$ and $y$ are not both synchronization actions $\wedge$
$x \in \text{prescient}_E \wedge y \notin \text{prescient}_E \wedge$
$x$ is not a write seen by $y$.

Figure 7: Definitions