Declarative semantics for concurrency

Ori Lahav Viktor Vafeiadis

28 August 2017

An alternative way of defining the semantics

Declarative/axiomatic concurrency semantics

- Define the notion of a program *execution*
  (generalization of an execution trace)
- Map a program to a set of executions
- Define a *consistency* predicate on executions
- Semantics = set of consistent executions of a program

Exception: "catch-fire" semantics

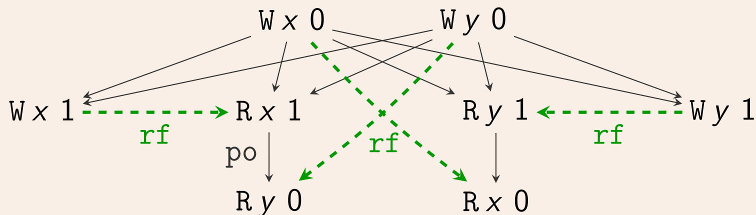
- Existence of at least one "bad" consistent execution implies undefined behavior.

Events

- Reads, Writes, Updates, Fences

Relations

- Program order, `po` (also called "sequenced-before", `sb`)
- Reads-from, `rf`



Definition (Label)

A *label* has one of of the following forms:

$$\mathtt{R}\,x\;\;v_r \qquad \mathtt{W}\,x\;\;v_w \qquad \mathtt{U}(x\;\;v_r\;\;v_w) \qquad \mathtt{F}$$

where $x \in \mathsf{Loc}$ and $v_r, v_w \in \mathsf{Val}$.

Definition (Event)

An *event* is a triple $\langle id, i, l \rangle$ where

- $id \in \mathbb{N}$ is an event identifier,
- $i \in \mathsf{Tid} \cup \{0\}$ is a thread identifier, and
- l is a label.

Definition (Execution graph)

An *execution graph* is a tuple $\langle E, po, rf \rangle$ where:

- E is a finite set of events
- po ("*program order*") is a partial order on E
- rf ("*reads-from*") is a binary relation on E such that:
  - For every $\langle w, r \rangle \in rf$
    - $\texttt{typ}(w) \in \{\texttt{W}, \texttt{U}\}$
    - $\texttt{typ}(r) \in \{\texttt{R}, \texttt{U}\}$
    - $\texttt{loc}(w) = \texttt{loc}(r)$
    - $\texttt{val}_{\texttt{w}}(w) = \texttt{val}_{\texttt{r}}(r)$
  - rf^{-1} is a function
    (that is: if $\langle w_1, r \rangle, \langle w_2, r \rangle \in rf$ then $w_1 = w_2$)

Some notations

Let $G = \langle E, po, rf \rangle$ be an execution graph.

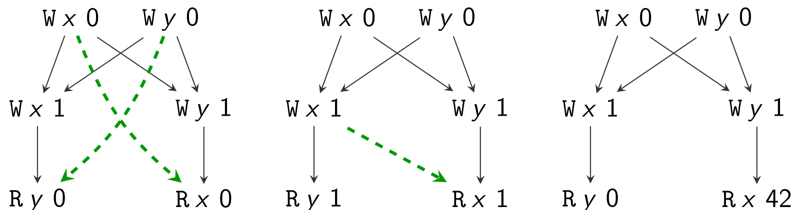
- $G.\mathtt{E} \triangleq E$
- $G.\mathtt{po} \triangleq po$
- $G.\mathtt{rf} \triangleq rf$
- $G.\mathtt{R} \triangleq \{r \in E \mid \mathtt{typ}(r) = \mathtt{R} \vee \mathtt{typ}(r) = \mathtt{U}\}$
- $G.\mathtt{W} \triangleq \{w \in E \mid \mathtt{typ}(w) = \mathtt{W} \vee \mathtt{typ}(w) = \mathtt{U}\}$
- $G.\mathtt{RMW} \triangleq \{u \in E \mid \mathtt{typ}(u) = \mathtt{U}\}$
- $G.\mathtt{F} \triangleq \{f \in E \mid \mathtt{typ}(f) = \mathtt{F}\}$
- $G.\mathtt{R}_x \triangleq G.\mathtt{R} \cap \{r \in E \mid \mathtt{loc}(r) = x\}$
- ...

Mapping programs to executions: Example

Store buffering (SB)

$$x = y = 0$$

$$\begin{array}{l||l} x := 1 & y := 1 \\ a := y & b := x \end{array}$$



Mapping programs to executions: Definition

- The thread subsystem associates a *sequential* execution graph to every command.
- A program execution is obtained by joining the sequential execution graphs of the constituent threads.

Definition

An execution graph G is called *sequential* if the following hold:

- $\mathtt{tid}(a) = 0$ for every $a \in G.\mathtt{E}$
- $G.\mathtt{po}$ is a total order on $G.\mathtt{E}$
- $G.\mathtt{rf} = \emptyset$

From commands to sequential execution graphs

Initial execution graph: $G_\emptyset$ - the empty graph

SILENT

$$\frac{c, s \xrightarrow{\varepsilon} c', s'}{c, s, G \Rightarrow c', s', G}$$

NON-SILENT

$$\frac{c, s \xrightarrow{l} c', s' \qquad l \neq \varepsilon \qquad a = \langle n, 0, l \rangle \qquad n \notin \{\mathtt{id}(b) \mid b \in G.\mathrm{E}\}}{c, s, G \Rightarrow c', s', \mathsf{Add}(G, a)}$$

where $\mathsf{Add}(G, a)$ is the execution graph G' given by:

- $G'.\mathrm{E} = G.\mathrm{E} \uplus \{a\}$
- $G'.\mathrm{po} = G.\mathrm{po} \cup (G.\mathrm{E} \times \{a\})$
- $G'.\mathtt{rf} = G.\mathtt{rf}$

Definition (Execution graph of a command)

G is a an execution graph of a command c with a final store s if $c, s_0, G_\emptyset \Rightarrow^* \textbf{skip}, s, G$.

Mapping programs to executions: Definition

Definition (Thread restriction)

Given $i \in \text{Tid}$ and an execution graph G, G^i denotes the sequential execution graph obtained by restricting G to the events $\{a \in G.\mathtt{E} \mid \mathtt{tid}(a) = i\}$, modifying their thread identifiers to 0, and discarding all $\mathtt{rf}$-edges.

Definition (Execution graph of a program)

G is an execution graph of a program P (with an outcome O) if G^i is an execution of $P(i)$ (with final store $O(i)$) for every $i \in \text{Tid}$.

Consistency predicate

Let X be some consistency predicate (on execution graphs)

Definition (Allowed outcome under a declarative model)

An outcome O is *allowed* for a program P under X if there exists an execution graph G such that:

- G is an execution graph of P with outcome O.
- G is X-consistent.

Exception: “catch-fire” semantics

... or if there exists an execution graph G such that:

- G is an execution graph of P.
- G is X-consistent.
- G is “bad”.

Completeness

The most basic consistency condition:

Definition (Completeness)

An execution graph G is called *complete* if

$$codom(G.\texttt{rf}) = G.\texttt{R}$$

i.e., *every* read reads from *some* write.

Sequential consistency

the result of any execution is the same as if the operations of all the processors were executed in some sequential order, respecting the order specified by the program [Lamport, 1979]

Sequential consistency [Lamport]

Definition

Let $\mathtt{sc}$ be a total order on $G.\mathtt{E}$. G is called SC-*consistent* wrt $\mathtt{sc}$ if the following hold:

- If $\langle a, b\rangle \in G.\mathrm{po}$ then $\langle a, b\rangle \in \mathtt{sc}$.
- If $\langle a, b\rangle \in G.\mathtt{rf}$ then $\langle a, b\rangle \in \mathtt{sc}$ and there does not exist $c \in G.\mathtt{W}_{\mathtt{loc}(b)}$ such that $\langle a, c\rangle \in \mathtt{sc}$ and $\langle c, b\rangle \in \mathtt{sc}$.

Definition

An execution graph G is called SC-*consistent* if the following hold:

- G is complete.
- G is SC-consistent wrt some total order $\mathtt{sc}$ on $G.\mathtt{E}$.

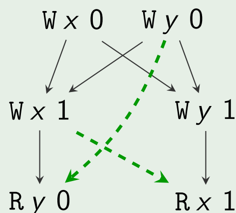
SB example

Store buffering (SB)

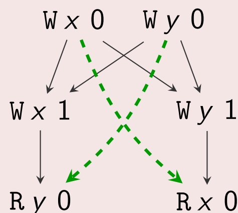
$$x = y = 0$$

$$\begin{array}{l} x := 1 \\ a := y \end{array} \;\Bigg\|\; \begin{array}{l} y := 1 \\ b := x \end{array}$$

Allowed



Forbidden



Sequential consistency (Alternative)

Definition (Modification order (aka coherence order))

$\mathtt{mo}$ is called a *modification order* for an execution graph G if $\mathtt{mo} = \bigcup_{x \in \mathsf{Loc}} \mathtt{mo}_x$ where each $\mathtt{mo}_x$ is a total order on $G.\mathtt{W}_x$.

Definition (Alternative SC definition)

An execution graph G is called SC-*consistent* if the following hold:

- G is complete
- There exists a modification order $\mathtt{mo}$ for G such that $G.\mathrm{po} \cup G.\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic where:
  - $\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathrm{id}$ (from-reads / reads-before)

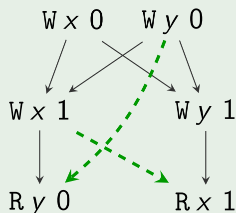
SB example

Store buffering (SB)

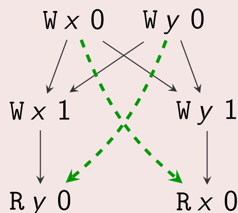
$$x = y = 0$$

$$\begin{array}{l} x := 1 \\ a := y \end{array} \;\Bigg\|\; \begin{array}{l} y := 1 \\ b := x \end{array}$$

Allowed



Forbidden



Equivalence

Theorem

The two SC definitions are equivalent.

Proof (sketch).

Lamport SC $\Rightarrow$ alternative SC:

- Take $\mathtt{mo}_x \triangleq [\mathtt{W}_x]; \mathtt{sc}; [\mathtt{W}_x]$.
- Then, $G.\mathrm{po} \cup G.\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb} \subseteq \mathtt{sc}$.

Alternative SC $\Rightarrow$ Lamport SC:

- Take $\mathtt{sc}$ to be any total order extending $G.\mathrm{po} \cup G.\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$. □

Relaxing sequential consistency

- SC is very expensive to implement in hardware.
- It also forbids various optimizations that are sound for sequential code.

What most hardware guarantee and compilers preserve is "SC-per-location" (aka *coherence*).

Definition

An execution graph G is called *coherent* if the following hold:

- G is complete
- For every location x, there exists a total order $\mathtt{sc}_x$ on all accesses to x such that:
  - If $\langle a, b\rangle \in [\mathtt{RW}_x]; G.\mathtt{po}; [\mathtt{RW}_x]$ then $\langle a, b\rangle \in \mathtt{sc}_x$
  - If $\langle a, b\rangle \in [\mathtt{W}_x]; G.\mathtt{rf}; [\mathtt{R}_x]$ then $\langle a, b\rangle \in \mathtt{sc}_x$ and there does not exist $c \in G.\mathtt{W}_x$ such that $\langle a, c\rangle \in \mathtt{sc}_x$ and $\langle c, b\rangle \in \mathtt{sc}_x$.

Alternative definition of coherence I

SC: $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
COH: $\mathtt{po}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic

Definition

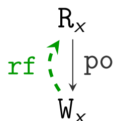
Let $\mathtt{mo}$ be a modification order for an execution graph G. G is called *coherent wrt* $\mathtt{mo}$ if $G.\mathtt{po}|_{\mathtt{loc}} \cup G.\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic (where $\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathrm{id}$).

Theorem

An execution graph G is coherent iff the following hold:

- *G is complete*
- *G is coherent wrt some modification order $\mathtt{mo}$ for G.*

“Bad patterns” I



$a := x$ // 1
$x := 1$

no-future-read

$r := \textbf{CAS}(x, 1, 1)$ // 1

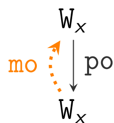
rmw-1

Recall:

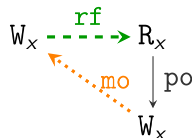
- $\mathtt{W}$ is either a write or an RMW.
- $\mathtt{R}$ is either a read or an RMW.

"Bad patterns" II

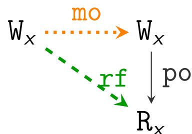
$$\begin{array}{l} x := 1 \\ x := 2 \end{array} \quad \Bigg\| \quad \begin{array}{l} a := x \ /\!/ \ 2 \\ a := x \ /\!/ \ 1 \end{array}$$



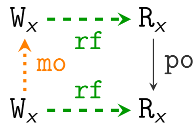
coherence-ww



coherence-rw

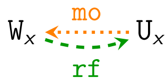


coherence-wr

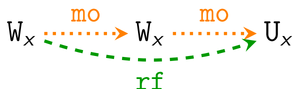


coherence-rr

"Bad patterns" III



rmw-2



atomicity

In coherent executions, an RMW event may only read from its immediate `mo`-predecessor.

Alternative definition of coherence II

Theorem

Let $\texttt{mo}$ *be a modification order for an execution graph* G.
G *is* *coherent* *wrt* $\texttt{mo}$ *iff the following hold:*

- $\texttt{rf}; \texttt{po}$ *is irreflexive.* *(no-future-read)*
- $\texttt{mo}; \texttt{po}$ *is irreflexive.* *(coherence-ww)*
- $\texttt{mo}; \texttt{rf}; \texttt{po}$ *is irreflexive.* *(coherence-rw)*
- $\texttt{rf}^{-1}; \texttt{mo}; \texttt{po}$ *is irreflexive.* *(coherence-wr)*
- $\texttt{rf}^{-1}; \texttt{mo}; \texttt{rf}; \texttt{po}$ *is irreflexive.* *(coherence-rr)*
- $\texttt{rf}$ *is irreflexive.* *(rmw-1)*
- $\texttt{mo}; \texttt{rf}$ *is irreflexive.* *(rmw-2)*
- $\texttt{rf}^{-1}; \texttt{mo}; \texttt{mo}$ *is irreflexive.* *(rmw-atomicity)*

Examples (aka "litmus tests")

Coherence test

$$x = 0$$

$$\begin{array}{l|l} x := 1 & x := 2 \\ a := x \;\; /\!/ \; 2 & b := x \;\; /\!/ \; 1 \end{array}$$

Store buffering

$$x = y = 0$$

$$\begin{array}{l|l} x := 1 & y := 1 \\ a := y \;\; /\!/ \; 0 & b := x \;\; /\!/ \; 0 \end{array}$$

Atomicity

Parallel increment

$$x = 0$$
$$a := \textbf{FAA}(x, 1) \parallel b := \textbf{FAA}(x, 1)$$

Guarantees that $a = 1 \lor b = 1$.

Can we implement locks in this semantics?

Spinlock implementation

$\underline{\textbf{lock}(l):}$

$r := 0;$
$\textbf{while } \neg r \textbf{ do}$
$\quad r := \textbf{CAS}(l, 0, 1)$

$\underline{\textbf{unlock}(l):}$

$l := 0$

Implementing locks?

Under COH, the spinlock implementation does not guarantee mutual exclusion.

Spinlock implementation

<u>**lock**(l) :</u>

$r := 0$;
while $\neg r$ **do**
  $r :=$ **CAS**$(l, 0, 1)$

<u>**unlock**(l) :</u>

$l := 0$

Lock example

lock(l)	**lock**(l)
$x := 1$	$y := 1$
$a := y$ // 0	$b := x$ // 0
unlock(l)	**unlock**(l)

Message passing

More generally, COH is often too weak:

$$x = y = 0$$

$$\begin{array}{l} x := 42; \\ y := 1 \end{array} \;\Bigg\|\; \begin{array}{l} a := y; \\ \textbf{while } \neg a \textbf{ do } a := y; \\ b := x \quad \textit{// 0} \end{array}$$

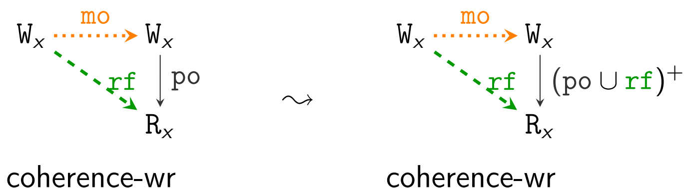
$$x = y = 0$$

$$\begin{array}{l} x := 42; \\ y := 1 \end{array} \;\Bigg\|\; \begin{array}{l} a := y; \quad \textit{// 1} \\ b := x \quad \textit{// 0} \end{array}$$

MP is a common programming idiom.

How can we disallow the weak behavior?

Supporting message passing



Solution:

- Strengthen the notion of an "observed" write.
- In other words, make rf-edges "synchronizing."

Release/acquire (RA) memory model

SC: $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
COH: $\mathtt{po}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
RA: $(\mathtt{po} \cup \mathtt{rf})^{+}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic

Definition

Let $\mathtt{mo}$ be a modification order for an execution graph G. G is called *RA-consistent wrt* $\mathtt{mo}$ if $(\mathtt{po} \cup \mathtt{rf})^{+}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic for some modification order $\mathtt{mo}$ for G (where $\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathrm{id}$).

Definition

An execution graph G is *RA-consistent* if the following hold:

- G is complete
- G is RA-consistent wrt some modification order $\mathtt{mo}$ for G.

Alternative definition of RA consistency

Theorem

Let $\texttt{mo}$ *be a modification order for an execution graph G. G is* RA-*consistent wrt* $\texttt{mo}$ *iff the following hold:*

- $(\mathrm{po} \cup \texttt{rf})^+$ *is irreflexive.* *(no-future-read)*
- $\texttt{mo}; (\mathrm{po} \cup \texttt{rf})^+$ *is irreflexive.* *(coherence-ww)*
- $\texttt{rf}^{-1}; \texttt{mo}; (\mathrm{po} \cup \texttt{rf})^+$ *is irreflexive.* *(coherence-wr)*
- $\texttt{rf}^{-1}; \texttt{mo}; \texttt{mo}$ *is irreflexive.* *(rmw-atomicity)*

Hardware implementation of RA

RA is cheaper to implement than SC, but some architectures still require some fences.

- On Power, a "lightweight" fence (`lwsync`) suffices, and RA is still cheaper than SC.
  - Release write $\rightsquigarrow$ `lwsync` ; store
  - Acquire read $\rightsquigarrow$ load ; `lwsync` *(or* load ; bc ; `isync`*)*
- ARMv7 is like Power, but has no lightweight fence.
  - Release write $\rightsquigarrow$ dmb ; store
  - Acquire read $\rightsquigarrow$ load ; dmb *(or* load ; bc ; isb*)*
- ARMv8 has special release/acquire accesses.
  - *Alternative:* acquire read $\rightsquigarrow$ load ; dmb ld
- On TSO, no fences are needed. (See also exercise.)

Mixing the models

$$\mathsf{COH} < \mathsf{RA} < \mathsf{SC}$$

- Revisit the MP idiom:

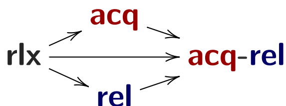
$$
\begin{array}{l||l}
x := 42 & a := y \\
y := 1 & \textbf{while } \neg a \textbf{ do } a := y \\
 & b := x \quad /\!/ 0
\end{array}
$$

- We only need the last read of y to synchronize.
- Idea: introduce *access modes*.

$$
\begin{array}{l||l}
 & a := y_{\mathsf{rlx}} \\
x :=_{\mathsf{rlx}} 42 & \textbf{while } \neg a \textbf{ do } a := y \\
y :=_{\mathsf{rel}} 1 & a := y_{\mathsf{acq}} \\
 & b := x_{\mathsf{rlx}} \quad /\!/ 0
\end{array}
$$

Happens-before

- Each memory accesses has a *mode*:
  - Reads: **rlx** or **acq**
  - Writes: **rlx** or **rel**
  - RMWs: **rlx**, **acq**, **rel** or **acq-rel**
- "Strength" order $\sqsubset$ is given by (the transitive closure of):



- Synchronization:

$$G.\mathtt{sw} = [\mathtt{W}^{\sqsupseteq \mathbf{rel}}]; G.\mathtt{rf}; [\mathtt{R}^{\sqsupseteq \mathbf{acq}}]$$

- Happens-before:

$$G.\mathtt{hb} = (G.\mathrm{po} \cup G.\mathtt{sw})^{+}$$

Towards C/C++11 memory model

SC: $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
COH: $\mathtt{po}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
RA: $(\mathtt{po} \cup \mathtt{rf})^{+}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic
C11: $\mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic

Definition

Let $\mathtt{mo}$ be a modification order for an execution graph G. G is called C11-*consistent wrt* $\mathtt{mo}$ if $\mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb}$ is acyclic (where $\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; \mathtt{mo} \setminus \mathrm{id}$).

Definition

An execution graph G is C11-consistent if the following hold:

- G is complete
- G is C11-consistent wrt some modification order $\mathtt{mo}$ for G.

The C/C++11 memory model

$\texttt{non-atomic} \sqsubset \texttt{relaxed} \sqsubset \texttt{release/acquire} \sqsubset \texttt{sc}$

The full C/C++11 is more general:

- Non-atomics for non-racy code (the default!)
- Four types of fences for fine grained control
- SC accesses to ensure sequential consistency if needed
- More elaborate definition of `sw` ("release sequences")

Summary

- A declarative approach for (weak) concurrency semantics
- Uniformly and modularly handle various models
- Important weakenings of SC (coherence, RA) with alternative formulations based on “bad patterns”
- Introduction to the C/C++11 memory model

Exercise: Extended coherence order

Let G be a coherent execution wrt some modification order $\texttt{mo}$ for G.

Let $\texttt{eco} \triangleq (\texttt{rf} \cup \texttt{mo} \cup \texttt{rb})^+$.

1. Does $\texttt{eco}$ totally order all accesses to a given location x?
2. Provide a simplification of $\texttt{eco}$ that avoids the use of transitive closure.

Exercise: RA vs. TSO

Do RA and TSO have the same behaviors?

1. Construct a program with two threads that has different outcomes under TSO and RA.
2. Construct a program without write-write races that distinguishes the two models.

Exercise: Write-before

Let G be a complete execution without RMW events and let:

$$\mathtt{wb} \triangleq [\mathtt{W}]; (\mathtt{po} \cup \mathtt{rf})^{+}|_{\mathtt{loc}}; [\mathtt{W}] \cup ([\mathtt{W}]; (\mathtt{po} \cup \mathtt{rf})^{+}|_{\mathtt{loc}}; \mathtt{rf}^{-1}; [\mathtt{W}] \setminus \mathrm{id})$$

1. Show that G is RA-consistent iff wb is acyclic.
2. (Optional, difficult) Extend the definition of wb to work for executions with RMW events.
3. (Optional, difficult) Can one define an analogue wb relation in terms of just G, such that G is SC-consistent iff wb is acyclic?