ECMAScript Axiomatic Memory Consistency Model

Akshay Gopalakrishnan

November 2019

0.1 Agents, Events and their Types

Agents A concurrent program involves different threads/processes running concurrently. Agents are analogous to different threads/processes. Agents actually have more meaning than what we refer to here. However, with respect to the memory consistency model , we can safely abstract them to just mean threads/processes.

Agent Cluster Collection of agents concurrently communicating with each other through means of shared memory form an agent cluster. There can be multiple agent clusters. However, an agent can only belong to one agent cluster. Agents communicating through message passing do not belong in the same agent cluster.

For our purpose, we assume just one agent cluster having one shared memory using which agents communicate.

Agent Event List (*ael*) Every agent is mapped to a list of events. Operationally, these events are appended to the list during evaluation. We define *ael* as a mapping of each agent to a list of events.

The standard refers this to be an Event List, but we find it a bit misleading as it does not signify a list for each agent. Hence we name it as Agent Event List.

0.2 Events

Agent execution is modelled in terms of events. An evaluation of an operation results in a set of events that are evaluated. An event is either an operation that involves (shared) memory access or that constrains the order of execution of multiple events.

Given an agent cluster, an *event set* $\boldsymbol{E}$ is a collection of all events from the agent event lists. This set is composed of mainly two distinct subsets as follows:

0.2.1 Shared Memory ($\boldsymbol{SM}$) Events

This set is composed of two sets of events:

1. Write events ($\boldsymbol{W}$) which write to shared memory.
2. Read events ($\boldsymbol{R}$) which read from shared memory.

Events that belong to both Write and Read events are called Read-Modify-Write.

0.2.2 Synchronize (S) Events

These events only restrict the ordering of execution of events by agents. For instance *lock* and *unlock* type of events canbe categorized under Synchronize events. However, this is not stated in the specification.

The features of $Lock$ and $Unlock$ events is actually not something given to the programmer to use in Javascript. They areused to implement the feature *wait* and $notify$ that the programmer can use which adhere to the semantics of $futexes$ inLinux. Hence, in the original standard of the model, the distinction between lock and unlock is not made, and it is simplystated as Synchronize Event.

There is an additional set of events called Host Specific Events, but for our purpose, it is not of any major concern.

Range ($\Re$) Each of the *shared memory events* are associated with a contiguous range of memory on which it operates. Range is afunction that maps a shared memory event to the range[1]it operates on. This we represent as a starting index i and a size$s$. So we could represent the range of a write event w as

$$\Re(w) = (i, s)$$

We define the two binary operators below on ranges:

1. Intersection ($\cap_{\Re}$) - Set of byte indices common to both ranges.
2. Union ($\cup_{\Re}$) - A unique set of byte indices that exist in both the ranges.

Two Ranges can be *disjoint*, *overlapping* or *equal*. We use the binary operators to define these threepossibilities between ranges of events e and d :

1. Disjoint $\Re(e) \cap_{\Re} \Re(d) = \phi$
2. Overlapping $(\Re(e) \cap_{\Re} \Re(d) \neq \phi) \wedge (\Re(e) \cap_{\Re} \Re(d) \neq \Re(e) \cup_{\Re} \Re(d))$ -
3. Equal $\Re(e) \cap_{\Re} \Re(d) = \Re(e) \cup_{\Re} \Re(d)$ - In simple terms, we define equality as $\Re(e) = \Re(d)$

0.2.3 Types of events based on Order

Order signifies the sequence in which event actions are visible to different agents as well as the order in which they are executed by the agents themselves. In our context, there are mainly three types (in C11 memory model, they are called access modes) for each shared memory event that tells us the kind of ordering that it enforces.

1. **Sequentially Consistent (sc)** - Events of this type are *atomic*[2]in nature. There is a strict global total ordering of such events which is agreed upon by all agents in the agent cluster.
2. **Unordered (uo)** - Events of this type are considered *non-atomic* and can occur in different orders for each concurrent process. There is no fixed global order respected by agents for such events.

[1]The range as per the ECMAScript standard denotes only the set of contiguous byte indices. The starting byte indexis kept separate. We find this to be unnecessary. Hence we define range to have starting index and size.

3. **Initialize (*init*)** - Events of this type are used to initialize the values in memory before events in an agent cluster begin to execute concurrently.

All events of type *init* are writes and all Read-Modify-Write events are of type *sc*. We represent the type of events in the memory consistency rules in the format "$event : type$". When representing events in examples, the type would be represented as a subscript: $event_{type}$.

0.2.4 Tearing (Or not)

Additionally, each shared-memory event is also associated with whether they are tear-free or not. OEvents that tear are non-aligned accesses requiring more than one memory access. Events that are tear-free are aligned and should appear to be serviced in one memory fetch[3].

0.3 Relation among events

We now describe a set of relations between events. These relations help us describe the consistency rules.

0.3.1 Read-Write event relations

There are two basic relations that assist us in reasoning about read and write events.

Read-Bytes-From ($\xrightarrow[rbf]{}$) This relation maps every read event to a list of tuples consisting of write event and their corresponding byte index that is read. For instance, consider a read event $r[i...(i+3)]$ and corresponding write events $w_1[i...(i+3)]$, $w_2[i...(i+4)]$. One possible $\xrightarrow[rbf]{}$ relation could be represented as

$$e \xrightarrow[rbf]{} \{(d1, i), (d2, i{+}1), (d2, i{+}2)\}$$

or having individual binary relation with each write-index pair as

$$e \xrightarrow[rbf]{} (d1, i),\; e \xrightarrow[rbf]{} (d2, i{+}1) \text{ and } e \xrightarrow[rbf]{} (d2, i{+}2).$$

Reads-From ($\xrightarrow[rf]{}$) This relation, is similar to the above relation, except that the byte index details are not involved in the composite list. So for the above example, the *rf* relation would be represented either as $e \xrightarrow[rf]{} (d1, d2)$ or individual binary read-write relation as $e \xrightarrow[rf]{} d1$ and $e \xrightarrow[rf]{} d2$. Figure below is an example of a program with its outcome (read values) shown in terms of reads-from relations.

[2]The word *atomic* does not imply the events are evaluated using just one instruction. For example, a 64-bit sequentially consistent write on a 32-bit system has to be done with two subsequent memory actions. But its intermediate state of write must not be seen by any other agent. In an abstract sense, this event must appear '*atomic*'.The *atomic* here also refers to implications of whether an event's consequence is visible to all other agents in the same global total order or not. The compiler must ensure that for each specific target hardware, such guarantees are satisfied.

[3]It is not clear whether the alignment is with respect to specific hardware or not. The notion of one memory fetch may not be possible for all hardware practically, but it is something that must appear so. We will see a rule for ensuring this in the memory consistency rules.

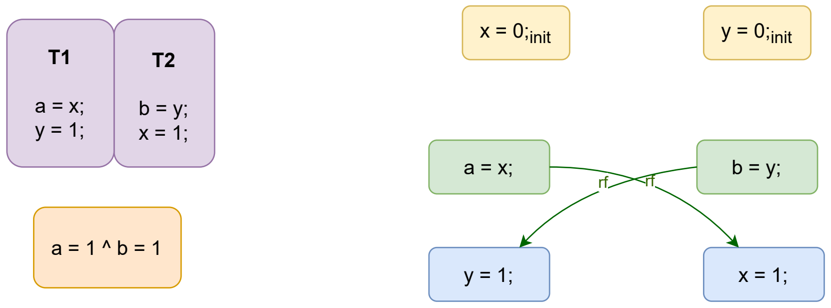

Figure 1: An example to show the reads-from relations that are drawn for the example program between read and write events.

0.3.2 Agent-Synchronizes With (ASW)

A list for each agent that consist of ordered tuples of synchronize events. These tuples specify ordering constraints among agents at different points of execution. So such a list for an agent k would be represented like:

$$ASW_k = \{\langle s_1, s_2\rangle, \langle s_3, s_4\rangle ...\}$$

For every pair in the list, the second event belongs to the parent agent and the first belongs to another agent it synchronized with[4].

$$\forall i, j > 0,\ \langle s_1, s_2\rangle \in ASW_j \Rightarrow s_2 \in ael(k)$$

The figure below shows an example of this relation among two agents.

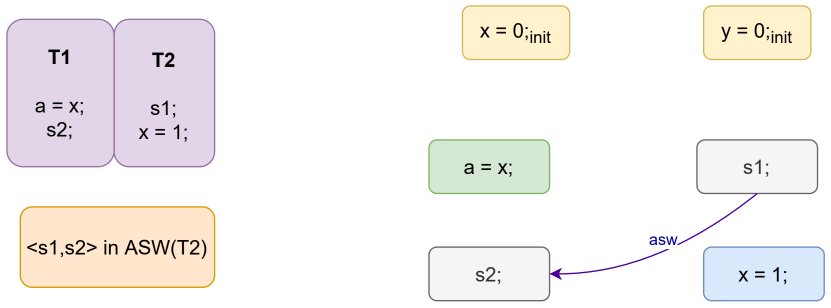

Figure 2: An example to show the reads-from relations that are drawn for the example program between read and write events.

[4]This is analogous to the property that every unlock must be paired with a subsequent lock, which enforces the condition that a lock can be acquired only when it has been released.

0.4 Ordering Relations among Events

0.4.1 Agent Order ($\xrightarrow[ao]{}$)

A total order among events belonging to the same agent event list. It is analogous to intra-thread ordering. For example, iftwo events e and d belong to the same agent event list , then either $e \xrightarrow[ao]{} d$ or $d \xrightarrow[ao]{} e$.

Note that the relations are only with respect to events belonging to the same agent. A collection of such relations togetherform the agent order.

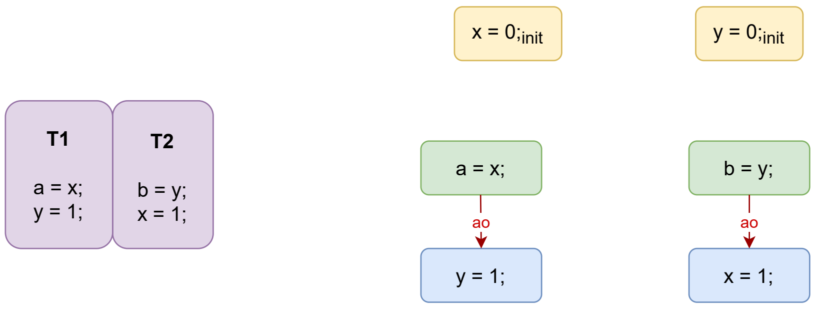

Figure 3: An example with agent order among the events.

0.4.2 Synchronize-With Order ($\xrightarrow[sw]{}$)

Represents the synchronizations among different agents through relations between their events. It is a composition of two setsas below:

1. All pairs belonging to ASW of every agent belongs to this ordering relation.

$$\forall i, j > 0,\ \langle e_i, e_j \rangle \in ASW \Rightarrow e_i \xrightarrow[sw]{} e_j$$

2. Specific reads-from pairs also belong to this ordering relation.

$$(r \xrightarrow[rf]{} w) \ \wedge\ r\!:\!sc \ \wedge\ w\!:\!sc \ \wedge\ (\Re(r)\!=\!\Re(w)) \ \Rightarrow\ (w \xrightarrow[sw]{} r)$$

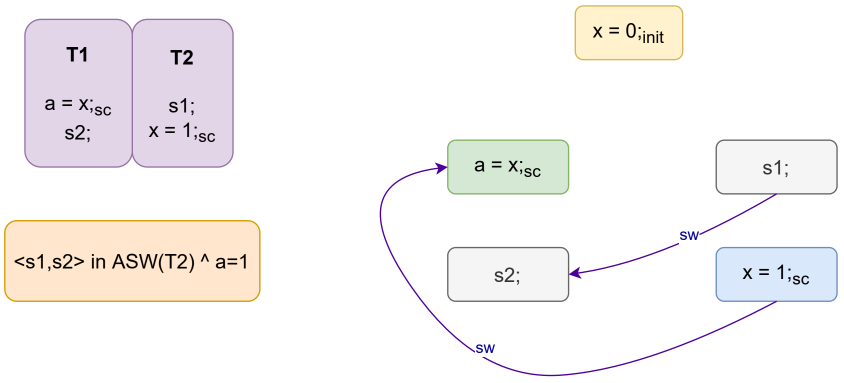

Figure 4: An example with synchronize with relations among the events.

Note that for the second condition, both ranges of events have to be equal. This however, does not mean thatthe read cannot read from multiple write events. (the read-from relation here is not functional.)

0.4.3 Happens Before Order ($\xrightarrow[hb]{}$)

A transitive order on events, composed of the following:

1. Every agent-ordered relation is also a happens-before relation

$$(e \xrightarrow[ao]{} d) \;\Rightarrow\; (e \xrightarrow[hb]{} d)$$

2. Every synchronize-with relation is also a happens-before relation

$$(e \xrightarrow[sw]{} d) \;\Rightarrow\; (e \xrightarrow[hb]{} d)$$

3. Initialize type of events happen before all shared memory events that have overlapping ranges with them.

$$\forall e, d \in SM \;\wedge\; e\!:\!init \;\wedge\; (\Re(e) \cap \Re(d) \neq \phi) \;\Rightarrow\; e \xrightarrow[hb]{} d$$

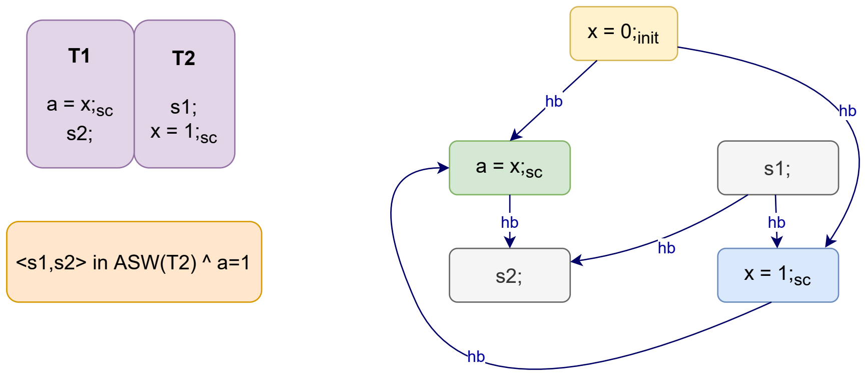

Figure 5: An example with all the types of happens-before relations between events.

0.4.4 Memory Order ($\xrightarrow[mo]{}$)

This order is a *total order* on all events that respects happens-before order.

$$e \xrightarrow[hb]{} d \Rightarrow e \xrightarrow[mo]{} d$$

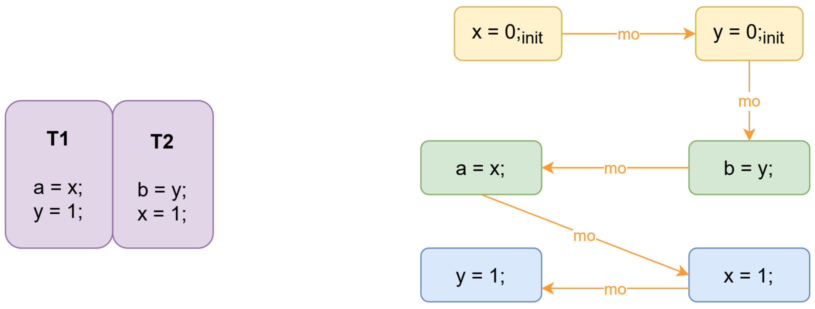

Figure 6: An example with a memory order (total) among all events.

An interesting part is that memory order, though total, is a bit undefined as to how it weaves together this total order given different init events. One can certainly make init events weaved among events that occur in each agent, thus making it sort of subjective as to when certain memory fragments are initialized. Discuss with Clark.

0.5 Preliminaries

Before we go into the consistency rules. we define certain preliminary definitions that create a separation based on a program, the axiomatic events and the various ordering relations defined above. This will help us understand where the consistency rules actually apply.

Definition 1. *Program A* program *is the source code without abstraction to a set of events and ordering relations. In our context, it is the original Javascript program.*

Definition 2. *Candidate This is a collection of abstracted set of shared memory events of a program involved in one possible execution, with the added* $\xrightarrow[ao]{}$ *relations. We can think of this as each thread having a set of shared memory events to run in a given intra-thread ordering. An example of a candidate is shown in figure 7.*

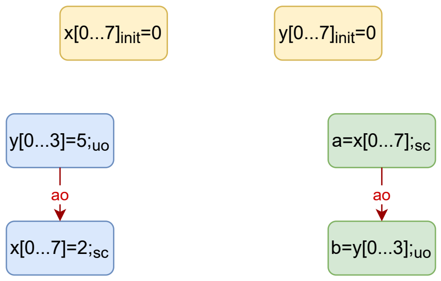

Figure 7: An example of a Candidate

Definition 3. *Candidate Execution A Candidate with the addition of $\xrightarrow[sw]{}$, $\xrightarrow[hb]{}$ and $\xrightarrow[mo]{}$ relations. This can be viewed as the witness/justification of an actual execution of a Program. Note that there can be many Candidate Executions for a given Candidate. The following figure shows an example of a candidate execution.*

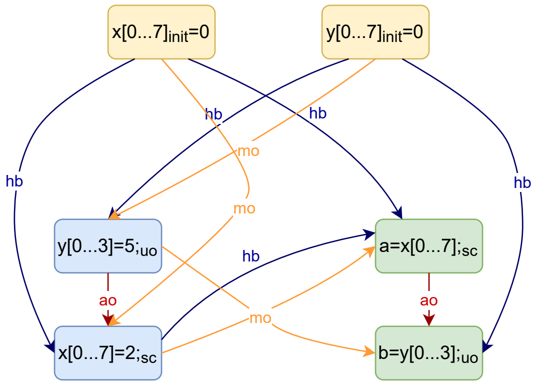

Figure 8: An example of an Execution based on Candidate above

Although by definition, the above relations are derived using $\xrightarrow[rf]{}$ relation, what we want to show is that given these relations exist, what are the implications on $\xrightarrow[rf]{}$ relations. Hence, our axioms of the memory model are based on restriction of $\xrightarrow[rf]{}$ contrast to it being restriction on these ordering relations that are adidtional in a Candidate Execution.

Definition 4. *Observable Behavior*

The set of pairwise $\xrightarrow[rf]{}$ and $\xrightarrow[rbf]{}$ relations that result in one execution of the program. Think of this as our outcome of a program execution.

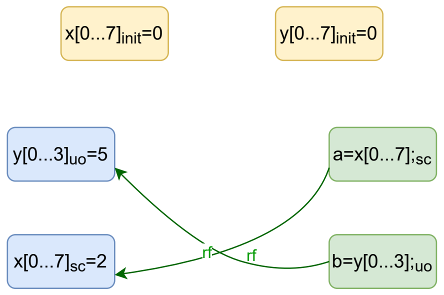

Figure 9: Observable Behavior

Make sure to change this figure to fit the modified definition of observable behaviors

The axioms of our memory model restrict the possible Observable Behaviors by specifying constraints on $\xrightarrow[rf]{}$ relations based on a Candidate Execution. For our purpose and flow in which we successively add relations to set of events, this would also include the implication on $\xrightarrow[rf]{}$ relation while having a $\xrightarrow[sw]{}$ relation among two events.

0.6 Valid Execution Rules (the Axioms)

We now state the memory consistency rules. The rules are on *Candidate Executions* which will place constraints on the possible *Observable behaviors* that may result from it.

0.6.1 Coherent Reads

There are certain restrictions of what a read event cannot see at different points of execution based on $\xrightarrow[hb]{}$ relation with write events.

Consider a read event e and a write event d having at least overlapping ranges:

$$e \in R \ \wedge \ d \in W \ \wedge \ (\Re(e) \cap_{\Re} \Re(d) \neq \phi).$$

- A read value cannot come from a write that has happened after it

$$e \xrightarrow[hb]{} d \ \Rightarrow \ \neg \ e \xrightarrow[rf]{} d.$$

  The figure below pictorially depicts the pattern above hwere e cannot read from d.

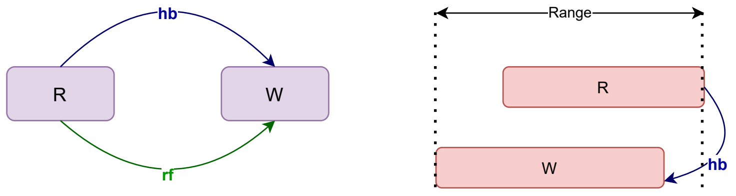

Figure 10: A read value cannot come from a write that has happened after it

- A read cannot read a specific byte address value from write if there is a write g that happens between them which modifies the exact byte address. Note that this rule would be on the rbf relation among two events.

$$d \xrightarrow[hb]{} e \ \wedge \ d \xrightarrow[hb]{} g \ \wedge \ g \xrightarrow[hb]{} e \ \Rightarrow \ \forall x \in (\Re(d) \cap_{\Re} \Re(g) \cap_{\Re} \Re(e)), \ \neg \ e \xrightarrow[rbf]{} (d, x).$$

  The figure below pictorially depicts the pattern where e cannot read certain bytes from d.

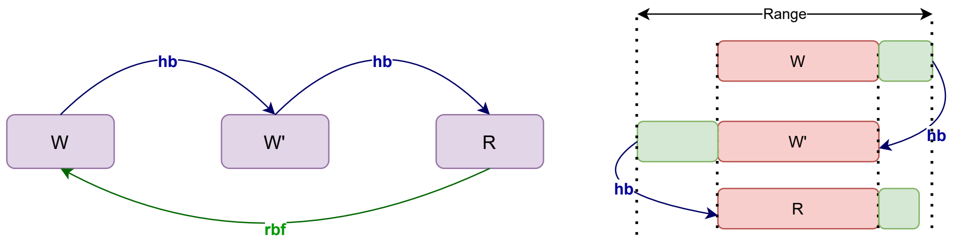

Figure 11: A read value cannot come from a write if there is a write that happens between them, writing to the same memory:

0.6.2 Tear-Free Reads

If two tear free writes d and g and a tear free read e all with equal ranges exist, then e can read only from one of them[5].

$$d : tf \wedge g : tf \wedge e : tf \wedge (\Re(d) = \Re(g) = \Re(e)) \Rightarrow ((e \xrightarrow[rf]{} d) \wedge (\neg\, e \xrightarrow[rf]{} g)) \vee ((e \xrightarrow[rf]{} g) \wedge (\neg\, e \xrightarrow[rf]{} d)).$$

0.6.3 Sequentially Consistent Atomics

To specifically define how events that are sequentially consistent affects what values a read cannot see, we assume the following memory order among writes d and g and a read e to be the premise for all the rules:

$$d \xrightarrow[mo]{} g \xrightarrow[mo]{} e.$$

- If all three events are of type sc with equal ranges, then e cannot read from d

$$d : sc \wedge g : sc \wedge e : sc \wedge (\Re(d) = \Re(g) = \Re(e)) \Rightarrow \neg\, e \xrightarrow[rf]{} d.$$

The figure below depicts pictorially the pattern that is not allowed by this rule.

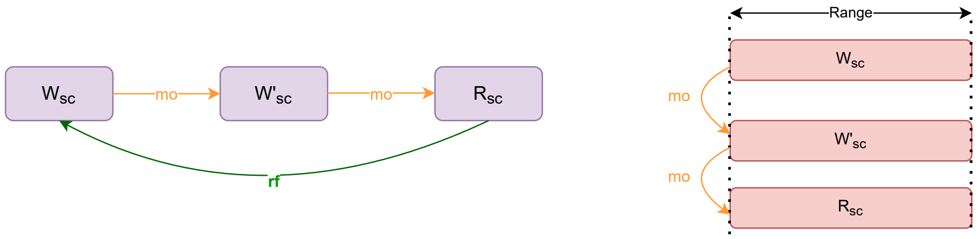

Figure 12: A read value cannot come from a write, if there exists a write memory ordered between them and all 3 events are sequentially consistent with equal ranges.

- If both writes are of type sc having equal ranges and the read is bound to happen after them, then e cannot read from d

$$d : sc \wedge g : sc \wedge (\Re(d) = \Re(g)) \wedge d \xrightarrow[hb]{} e \wedge g \xrightarrow[hb]{} e \Rightarrow \neg\, e \xrightarrow[rf]{} d.$$

The figure below depicts pictorially the pattern that is not allowed by this rule.

[5]To recap a tear-free event cannot be separated into multiple small events that do the same operation. However, considering different hardware architectures, the notion of tear-free need not necessarily mean this. (eg: A 64bit tear-free write to be done in a 32bit system). In a more abstract sense, we need an event to appear 'tear-free'.

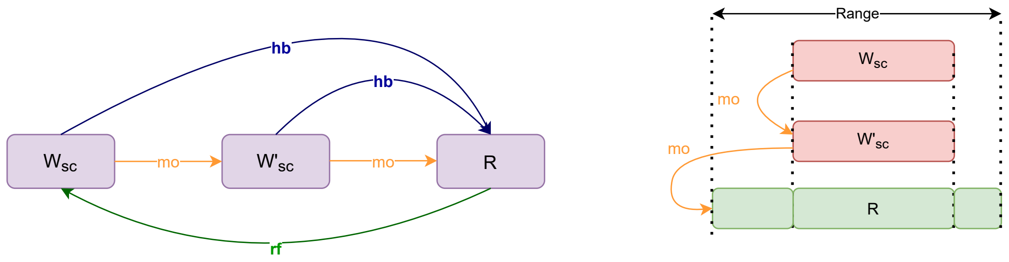

Figure 13: A read value cannot come from a write, if there exists a write memory ordered between them and both writes are sequentially consistent with equal ranges.

- If g and e are sequentially consistent, having equal ranges, and d is bound to happen before them, then e cannot read from d

$$g\!:\!sc \ \wedge \ e\!:\!sc \ \wedge \ (\Re(g)\!=\!\Re(e)) \ \wedge \ d \xrightarrow[hb]{} g \ \wedge \ d \xrightarrow[hb]{} e \ \Rightarrow \ \neg \ e \xrightarrow[rf]{} d.$$

The figure below depicts pictorially the pattern that is not allowed by this rule.

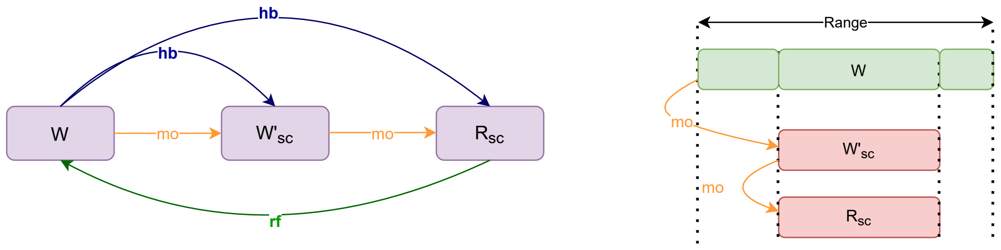

Figure 14: A read value cannot come from a write, if there exists a write memory ordered between them and both this write and the read are sequentially consistent with equal ranges.

0.7 Race

0.7.1 Race Condition RC

We define $\boldsymbol{RC}$ as the set of all pairs of events that are in a race. Two events e and d are in a race condition when they are shared memory events:

$$(e \in SM) \ \wedge \ (d \in SM).$$

having overlapping ranges, not having a $\xrightarrow[hb]{}$ relation with each other, and which are either two writes or the two events are involved in a $\xrightarrow[rf]{}$ relation with each other. This can be stated concisely as,

$$\neg \ (e \xrightarrow[hb]{} d) \ \wedge \ \neg \ (d \xrightarrow[hb]{} e) \ \wedge \ ((e, d \!\in\! W \ \wedge \ (\Re(d) \cap_{\Re} \Re(e) \neq \phi)) \ \vee \ (d \xrightarrow[rf]{} e) \ \vee \ (e \xrightarrow[rf]{} d)).$$

0.7.2 Data Race DR

We define ***DR*** as the set of all pairs of events that are in a data-race. Two events are in a data race when they are already in a race condition and when the two events are not both of type sc, or they have overlapping ranges. This is concisely stated as:

$$e, d \in RC \ \wedge \ ((\neg e : sc \ \vee \ \neg d : sc) \ \vee \ (\Re(e) \cap_{\Re} \Re(d) \neq \Re(e) \cup_{\Re} \Re(d)))$$

The definition for data race also implies that sequentially consistent events with overlapping ranges are also in a data race. This may be counter-intuitive in the sense that all agents observe the same order in which these events happen.

Data-Race-Free (DRF) Programs An execution is considered data-race free if none of the above conditions for data-races occur among events. A program is data-race free if all its executions are data race free. *The memory model guarantees Sequential Consistency for all data-race free programs (SC-DRF).*

0.8 Consistent Executions (Valid Observables)

A valid observable behaviour is when[6]:

1. No $\xrightarrow[rf]{}$ relation violates the above memory consistency rules.
2. $\xrightarrow[hb]{}$ is a strict partial order.

The memory model guarantees that every program must have at least one valid observable behaviour.

[6]There is also some conditions on host-specific events (which we mentioned is not of our main concern) and what is called a chosen read, which is nothing but the reads that the underlying hardware memory model allows. Since we are not concerned with the memory models of different hardware, this restriction on reads is not of our concern.