ECMAScript Axiomatic Memory Consistency Model

Akshay Gopalakrishnan

November 2019

1 Agents, Events and their Types

1.1 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, in terms of reasoning just with memory consistency rules, we are safely abstracting them to just mean threads/processes.

Agent Cluster Collection of agents concurrently communicating with each other (directly/ indirectly) form an agent cluster. There can be multiple agent clusters. However, an agent can only belong to one agent cluster.

Please look back at what Conrad had said to about agent clusters.

Note that for the purpose of reasoning with optimizations given the memory model, we stick to assuming that just one agent cluster exists. We also assume that agents in the cluster communicate only through one common shared memory segment.

Could elaborate the role when different shared array buffers are used to communicate accross agents belonging to different clusters. However, this is something that is not primary to our purpose of investigation, but would be essential as a whole from a practical standpoint to enforce correct concurrent programming.

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

$$ael(a) = [e_1, e_2, ...e_k]$$

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

1.2 Events

The memory model is described mainly using a set of events and some ordering relations on them. 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. The latter are called *Synchronize Events*

Synchronizing events are analogous to *lock* and *unlock* events that allow exclusive access to critical sections of memory. However, this is not specified in the standard as part of the memory model.

1.3 Event Set

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

1.3.1 Shared Memory (SM) Events

This set is composed of two sets of events:

Use a better listing to enumerate both items below in the same line

1. Write events ($\boldsymbol{W}$)
2. Read events ($\boldsymbol{R}$)

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

1.3.2 Synchronize (S) Events

These events only restrict the ordering of execution of events by agents. They are of two sets, which are mutually exclusive:

1. Lock events ($\boldsymbol{L}$)
2. Unlock events ($\boldsymbol{U}$)

The features of *Lock* and *Unlock* events is actually not something given to the programmer to use in Javascript. They are used to implement the feature *wait* and *notify* that the programmer can use which adhere to the semantics of *futexes* in Linux. Hence, in the original standard of the model, the distinction between lock and unlock is not made, and it is simply stated 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 a function that maps a shared memory event to the range 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

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

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

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 three possibilities 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 $\Re(e) = \Re(d)$

Note that two ranges being overlapping is different from them being equal. This distinction is used to define certain things ahead in the model.

Value(V) It is a function that maps a byte address given to the value that is stored in that address.For example, the byte address k has the value x_k will be depicted as:

$$V(k) = x_k$$

We introduce the value function to just map memory to values stored there. Note that we also assume only integer values for the sake of reasoning with memory models.

Consider when and where these notations are used, and if not early as this, refrain from mentioning it here.

Using the above constructs, we represent the three subset of shared memory events with their ranges in the following way:

Consider a chunk of memory k,k+1...k+10 wherein the values stored are:

$$\forall i \in [0, 10], V(k + i) = x_{k+i}$$

- W with range $(k, 11)$ modifying memory to $x'_k ... x'_{k+10}$ will be as :

$$W^i_j[k...(k+10)] = \{x'_k, x'_{k+1}...x'_{k+10}\}$$

- R will be represented the same as write with a distinction in semantics that the right hand side is what is read from the range of memory

$$R^i_j[k...(k+10)] = \{x_k, x_{k+1}...x_{k+10}\}$$

- RMW will be mapped to two tuples, the left one indicating the values read and the right one indicating the values written to the same memory.

$$RMW^i_j[k...(k+10)] = \{(x_k, x_{k+1}...x_{k+10}), ({x'}_k, {x'}_{k+1}...{x'}_{k+10})\}$$

Note that some examples will also be like $R[0..4] = 10$, where 10 symbolizes the value stored in 32 bits of memory, which is ideally the form $\{0, 0, 0, 10\}$. This is because, we are taking decimal equivalent of a 32 bit binary number.

1.4 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 literature of C11, called as 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* 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.
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.

Verify whether the set of agents that agrees upon tot of SC are the ones in the same agent cluster or those that share the same memory.

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: $event_{type}$.

The word *atomic* is actually misleading. It 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.

The notion of sequentially consistent has the same semantics of what C11 has for such events. This is gained through discussion with a few who were instrumental in designinig this model. However, it must be checked whether the semantics does indeed mimic that of C11

It is unclear from the standard if *init* is a type of write that has a range as the range of shared memory involved in the agent cluster or is it individual writes for each byte address. This is important as it plays a role in establishing certain ordering constraints on events.

1.5 Tearing (Or not)

Additionally, each shared-memory event is also associated with whether they are tear-free or not. Operations that tear are not aligned accesses and can be serviced using two or more memory fetches. Operations that are tear-free are aligned and should appear to be serviced in one memory fetch.

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.

There is a very confusing definition of *tear-free ness* given by ECMAScript. These definitions are part of how the tear factor affects the behavior of programs in a concurrent setting:

1. For every Read event, tear-free-ness questions whether this event is allowed to read from multiple write events on equal range as this event
2. For every Write event, tear-free-ness questions whether this event is allowed to be read by multiple reads on equal range as this event.

This needs to be discussed perhaps.

2 Relation among events

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

2.0.1 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 $e \xrightarrow[rf]{} d2$.

2.0.2 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.

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

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.

3 Ordering Relations among Events

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, if two 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 together form the agent order. This is analogous to what we know as intra-thread sequential order. It is also the same as what **sequenced-before** is defined to be in C++.

Sequenced before maybe a bit weaker that agent order, as we saw in one of the papers. Basically, sequenced before precisely tells us which events can be reordered and which cannot, in contrast to agent order. I think it is similar to preserved program order in hardware, which is defined to capture dependancies resulting due to control flow in programs. Do check and see if these are the same. Discuss with Clark.

Synchronize-With Order ($\xrightarrow[sw]{}$) Represents the synchronizations among different agents through relations between their events. It is a composition of two sets as 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.

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

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

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.

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

It is also important to note that those $\xrightarrow[hb]{}$ relations that are formed due to Sequentially Consistent events (read-write), imply a more stronger visibility guarantee, in that all the threads observe the same global total order of such events. This however, is not expressed using this relation. Perhaps a better way to represent it may be required.

Memory Order ($\xrightarrow[mo]{}$) This order is a *total order* on all events that respects happens-before order.

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

3.1 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 1.*

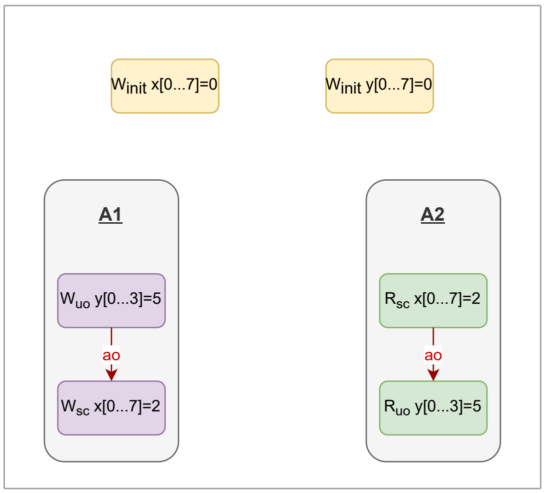


Figure 1: An example of a Candidate

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.*

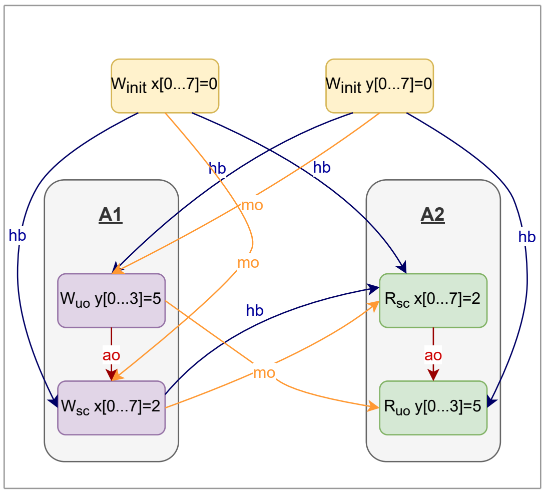


Figure 2: An example of an Execution based on Candidate above

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.

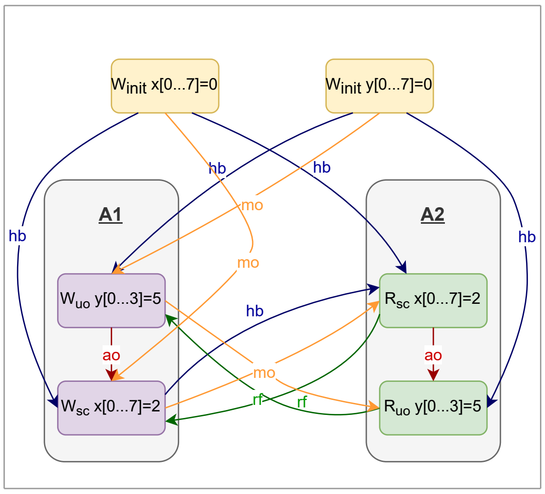


Figure 3: Observable Behavior

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.

4 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.

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

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

- 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.

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

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

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

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'.

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

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

- 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

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

- 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 standard specification talks of this in terms of what sequentially consistent write g should not be there when an $\xrightarrow[rf]{}$ relation exists among two events. We however, describe it in terms of disallowed $\xrightarrow[rf]{}$ relation to keep the rules consistent

We think we do not necessarily need ranges to be equal in some cases, however, this needs to be looked into more carefully.

Write events that are sequentially consistent are observed to happen in the same memory order by every agent. This is without any specific $\xrightarrow[hb]{}$ relation among such events. Does this really hold ? This needs to be discussed separately.

5 Race

Race Condition *RC* We define ***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,

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

Though we say it as write events, they also encompass read-modify-write events, as specified by the axiom above.

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).*

6 Consistent Executions (Valid Observables)

A valid observable behaviour is when:

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.

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.

7 Instruction Reordering

Instruction reordering is a common operation in compiler optimization, essential to instruction scheduling of course, but also implicit in loop invariant removal, partial redundancy elimination, and other optimizations that may move instructions. However, whether we can do such reordering freely given a concurrent program using relaxed memory accesses is a bit unclear.

Simple reordering is not straightforward under shared memory semantics The main reason is that memory accesses here, do not just perform the desired operation (i.e Read / Write) but also imply certain visibility guarantees across all the other threads. In our observation, we find that, the relaxed memory model of Javascript prescribe semantics for visibility using the $\xrightarrow[hb]{}$ relations.

Show an example or multiple examples here that enforces visibility due to having sequentially consistent events involved in a Candidate Execution.

What can be done? An example-based analysis exposes to us the problems that might exist when we perform such reordering of events. However, such an analysis, though would work for small programs to identify the possible conditions under which reordering can be done, become infeasible as the programs scale in length and complexity. This is because of the exponential increase in possible executions as the number of threads and program size in general increase. Hence, generalizations by using a small sample size is not something we can afford especially when we want to ensure these program trasnformations are done by the compiler in contrast to being done manually.

Our approach Our solution to this is to construct a proof on Candidate Executions of the original program and the transformed one which exposes the possible observable behaviors it can have. The crux of the proof is to guarantee that reordering does not bring any new $\xrightarrow[rf]{}$ (reads-from) relations that did not exist in any Observable Behavior of the original Candidate Execution. It is important to note however, that a proof in this sense would be generalized to any Candidate and is thus conservative. So, it might be the case that for specific programs, reordering can be valid, however, in a general sense may not be valid for others.

Assumption We make the following assumptions for every program we consider :

1. All events are tear-free
2. No synchronize events exist
3. No Read-Modify-Write events exist
4. All executions of the candidate before reordering have happens-before as a strict partial order

We first consider when consecutive events in the same agent can be reordered, followed by non-consecutive cases. The crux of the proof is to guarantee that reordering does not bring any new reads-from relations that did not result due to any execution of the original program.

7.1 Preliminaries

Before we go about proving when reordering is valid, we would like to have two additional definitions which would prove useful.

Definition 5. *Consecutive pair of events (*cons*)*

We define cons *as a function, which takes two events as input, and gives us a boolean indicating if they are consecutive pairs. Two events e and d are consecutive if they have an $\xrightarrow[ao]{}$ relation among them and are* next to each other*, which can be defined formally as*

$$(e \xrightarrow[ao]{} d \ \wedge \ \nexists k \ s.t. \ e \xrightarrow[ao]{} k \ \wedge \ k \xrightarrow[ao]{} d) \ \vee \ (d \xrightarrow[ao]{} e \ \wedge \ \nexists k \ s.t. \ d \xrightarrow[ao]{} k \ \wedge \ k \xrightarrow[ao]{} e)$$

Definition 6. *Direct happens-before relation (dir)*

We define dir *to take an ordered pair of events (e, d) such that $e \xrightarrow[hb]{} d$ and gives a boolean value to indicate whether this relation is direct, i.e those relations that are not derived through transitive property of $\xrightarrow[hb]{}$.*

We can infer certain things using this function based on some information on events e and d.

- *If $e\!:\!uo$, then $dir(e, d) \ \Rightarrow \ cons(e, d)$*
- *If $d\!:\!uo$, then $dir(e, d) \ \Rightarrow \ cons(e, d)$*
- *If $e\!:\!sc \ \wedge \ e\!\in\! R$, then $dir(e, d) \ \Rightarrow \ cons(e, d)$*
- *If $e\!:\!sc \ \wedge \ e\!\in\! W$, then $dir(e, d) \ \Rightarrow \ cons(e, d) \ \vee \ e \xrightarrow[sw]{} d$*
- *If $d\!:\!sc \ \wedge \ d\!\in\! W$, then $dir(e, d) \ \Rightarrow \ cons(e, d)$*
- *If $d\!:\!sc \ \wedge \ e\!\in\! R$, then $dir(e, d) \ \Rightarrow \ cons(e, d) \ \vee \ e \xrightarrow[sw]{} d$*

7.2 Lemmas to assist our proof

In order to assist our proof, we define two *lemmas* based on the ordering relations.

Lemma 1. *Consider three events e,d and k.*

If

$$cons(e, d) \ \wedge \ e \xrightarrow[ao]{} d \ \wedge \ ((d\!:\!uo) \ \vee \ (d\!:\!sc \ \wedge \ d\!\in\! W))$$

then,

$$k \xrightarrow[hb]{} d \Longrightarrow k \xrightarrow[hb]{} e$$

When we have two consecutive events e and d which are one after the other (i.e. $e \xrightarrow[ao]{} d$), we can use transitive property of $\xrightarrow[hb]{}$ to infer that any event k that happens before e, also happens before d. However, is it possible to derive that the event k happens before e using the evidence that k happens before d ? This lemma states the condition when this is true.

Proof. We will divide the proof for this into two cases, based on what event d is. For both cases, we have the following to be true :

$$cons(e, d) \ \wedge \ e \xrightarrow[ao]{} d \tag{0}$$

In the first case,

$$d\!:\!uo \tag{1}$$

Then for any event k

$$dir(k, d) \Rightarrow cons(k, d) \qquad from \quad (1) \tag{2}$$

An event that satisfies the above with d is e.

$$k = e \qquad from \quad (0, 2) \tag{3}$$

Because $\xrightarrow[ao]{}$ is a total order, e will be the only event. This would mean that for any other $k \neq e$,

$$k \xrightarrow[hb]{} d \Rightarrow k \xrightarrow[hb]{} d \qquad from \quad (0, 1, 2, 3)$$

The following figure should explain this intuition:

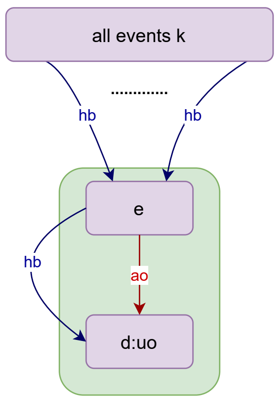


Figure 4: For the first case

In the second case,

$$d\!:\!sc \wedge d \in W \tag{4}$$

Then for any event k

$$dir(k,d) \Rightarrow cons(k,d) \qquad from \quad (4) \tag{5}$$

We once again have event e satisfying the above

$$k = e \qquad from \quad (0,5) \tag{6}$$

Though there could be direct *happens-before* relation with some event k from another *agent*, these are only relations satisfying $dir(d,k)$. Thus, we can once again infer that for any $k \neq e$

$$k \underset{hb}{\longrightarrow} d \Rightarrow k \underset{hb}{\longrightarrow} d \qquad from \quad (0,4,5,6)$$

The following figure explains this intuition:

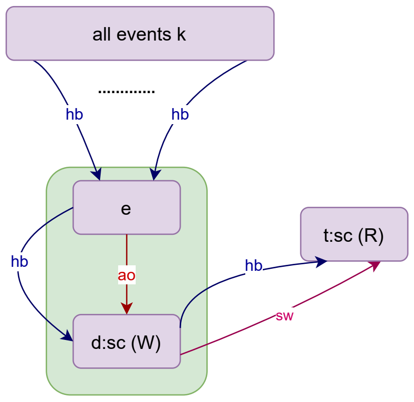


Figure 5: For the second case

□

Lemma 2. *Consider three events e, d and k*

If

$$cons(e,d) \ \wedge \ e \underset{ao}{\longrightarrow} d \ \wedge \ ((e\!:\!uo) \ \vee \ (e\!:\!sc \ \wedge \ e \in R))$$

then,

$$e \underset{hb}{\longrightarrow} k \Longrightarrow d \underset{hb}{\longrightarrow} k$$

When we have two consecutive events e and d which are one after the other (i.e. $e \xrightarrow[ao]{} d$), we can use transitive property of $\xrightarrow[hb]{}$ to infer that any event k that happens after d, also happens after e. However, is it possible to derive that the event k happens after d using the evidence that k happens after e ? This lemma states the condition when this is true.

Proof. Just like the proof for the previous lemma, we will divide the proof for this into two cases, based on what event e is. Again, for both cases, we have the following to be true:

$$cons(e,d)\ \land e \xrightarrow[ao]{} d \tag{0}$$

In the first case,

$$e : uo \tag{1}$$

Then for any event k

$$dir(e,k) \Rightarrow cons(e,k) \qquad from \quad (1) \tag{2}$$

The event that satisfies the above with e is d

$$k = d \qquad from \quad (0,2) \tag{3}$$

Because $\xrightarrow[ao]{}$ is a total order, d would be the only such event. This would mean that for any other event $k \neq d$

$$e \xrightarrow[hb]{} k \Rightarrow d \xrightarrow[hb]{} k \qquad from \quad (0,1,2,3)$$

The following figure should explain this intuition:

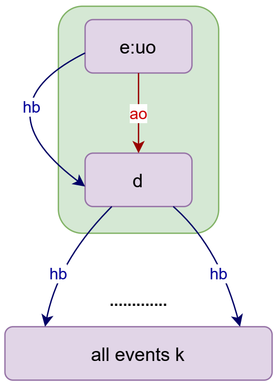


Figure 6: Caption

In the second case,

$$e : sc \land e \in R \tag{4}$$

Then for any event k

$$dir(e,k) \Rightarrow cons(e,k) \qquad from \quad (4) \tag{5}$$

We once again have event d satisfying the above

$$k = d \qquad from \quad (0,5) \tag{6}$$

Though there could be direct *happens-before* relation with some event k from another *agent*, these are only relations satisfying $dir(k,e)$. Thus, we can once again infer that for any $k \neq d$

$$e \xrightarrow[hb]{} k \Rightarrow d \xrightarrow[hb]{} k \qquad from \quad (0,4,5,6)$$

The following figure explains this intuition:

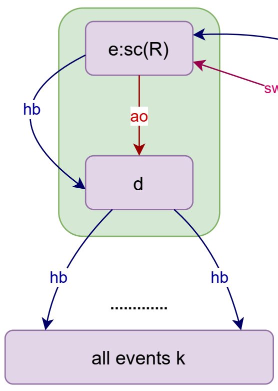


Figure 7: Caption

□

7.3 Valid reordering

We view reordering as manipulating the agent-order relation among two events. In that sense, reordering two consecutive events e and d such that $e \xrightarrow[ao]{} d$ becomes:

$$e \xrightarrow[ao]{} d \longmapsto d \xrightarrow[ao]{} e$$

What implications this change has on the other ordering relations depends on the type of events e and d are and would require an analysis on each Candidate Execution. The intuition is that the axioms of the memory model rely on certain ordering relations to restrict observable behaviors in a program. Hence, preserving these ordering relations would help us in turn not introduce new Observable Behaviors. In particular we note that preserving $\xrightarrow[hb]{}$ relations (other than the one we eliminate intentionally i.e $e \xrightarrow[hb]{} d$) would suffice for our purpose. Since $\xrightarrow[mo]{}$ respects $\xrightarrow[hb]{}$, we in turn even preserve the memory order which is essential.

In the end, we want to ensure that the set of possible observable behaviors of a program, remain unchanged after reordering. If that is not feasible, then we would want the set of observable behaviors after reordering at the very least to be a subset. This would ensure that the program does not have some new behaviours that weren't supposed to happen prior to reordering.

We begin by first defining a reorderable pair of events. We then formulate a theorem (with a proof) on the set of observable behaviors of a Candidate before and after reordering a pair of consecutive events which are reorderable. We consider reordering valid if the set of observable behaviours after reordering are a subset of the original.

Definition 7. *Reorderable Pair (Reord) We define a boolean function* Reord *that takes two ordered pair of events e and d such that $e \xrightarrow[ao]{} d$ and gives a boolean value indicating if they are a reorderable pair.*

$$
\begin{aligned}
&Reord(e,d) = \\
&\qquad (((e{:}uo \;\wedge\; d{:}uo) \;\wedge\; ((e{\in}R \;\wedge\; d{\in}R) \;\vee\; (\Re(e) \cap_{\Re} \Re(d) = \phi))) \\
&\qquad \vee \\
&\qquad ((e{:}sc \;\wedge\; d{:}uo) \;\wedge\; ((e{\in}W \;\wedge\; (\Re(e) \cap_{\Re} \Re(d) = \phi)))) \\
&\qquad \vee \\
&\qquad ((e{:}uo \;\wedge\; d{:}sc) \;\wedge\; ((d{\in}R \;\wedge\; (\Re(e) \cap_{\Re} \Re(d) = \phi)))))
\end{aligned}
$$

Use the latter for the purpose at the end of the proof for reordering, to emphasize how we approached each case

Theorem 2.1. *Consider a candidate C of a program and its possible Candidate Executions where $\xrightarrow[hb]{}$ is strictly partial order. Consider two events e and d such that cons(e, d) is true in C and $e \xrightarrow[ao]{} d$. Consider another candidate C' resulting after reordering e and d. Then if* Reord(e,d) *is true in C, the set observable behaviors possible due to Candidate Executions of C' is a subset of that of C.*

Proof. We look at this in terms of performing an instruction reordering on a candidate execution of C. We would want the resulting candidate execution to preserve all the other $\xrightarrow[hb]{}$ relations (except $e \xrightarrow[hb]{} d$) and that any new $\xrightarrow[hb]{}$ relations strictly reduce possible observable behaviors.

The proof is structured as follows. We first show that existing *happens-before* relations in any candidate execution of C except $e \xrightarrow[hb]{} d$ remain intact after reordering. We then identify the cases where new *happens-before* relations could be established. We identify from these cases whether *happens-before* cycles could be introduced. We then show for the remaining cases that new relations do not introduce any new observable behaviors.

The above steps can be summarized as addressing four main questions for any $CandidateExecution$ of C'

1. Apart from $e \xrightarrow[hb]{} d$, do other *happens-before* relations remain intact?
2. Apart from $d \xrightarrow[hb]{} e$, are any new *happens-before* relations established?
3. Are any *happens-before* cycles introduced?
4. Do the new relations bring new *observable behaviors?*

1. Preserving *happens-before* relations If some $\xrightarrow[hb]{}$ relations among events are lost after reordering, we may introduce new observable behaviors.

The relations that could be subject to change can be addressed by considering two disjoint sets of events in any *Candidate Execution* of C as below.

$$K_e = \{k \mid k \xrightarrow[hb]{} e\}.$$
$$K_d = \{k \mid d \xrightarrow[hb]{} k\}.$$

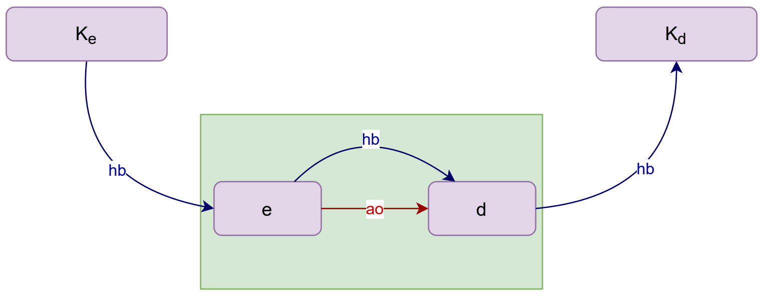


Figure 8: For any Candidate Execution of C, the set K_e and K_d

The idea is that if these relations are intact, then the relations among events from the first and second set will also hold due to transitivity.

Consider two events $p1 \in K_e$ and $p2 \in K_d$ (When e is the first event or d is the last event, assume dummy events that can act as $p1$ or $p2$.) belonging to the same agent as that of e and d such that in C:

$$dir(p1, e) \ \wedge \ dir(d, p2).$$

Note that in terms of direct happens-before relations, on reordering, any $CandidateExecution$ of C will have the following changes

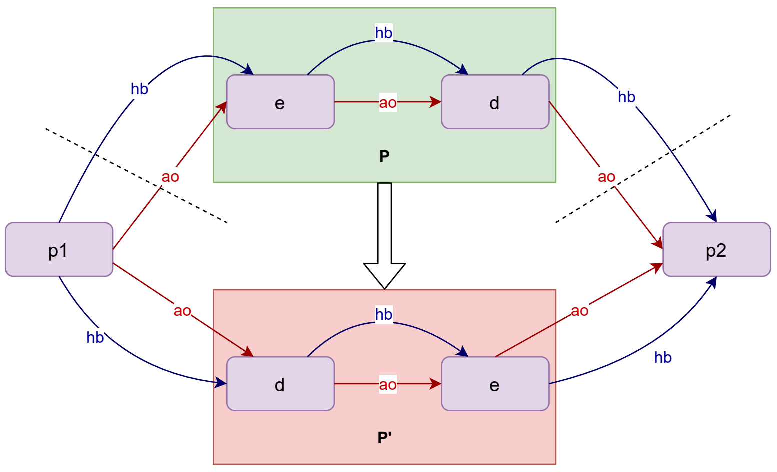


Figure 9: The direct relation changes that can be observed while reordering events e and d

The figure above is to show that, for any $CandidateExecution$ of C, the following is true

$$cons(p1, e) \ \wedge dir(p1, e) \ \wedge dir(e, d) \ \wedge cons(d, p2) \ \wedge \ dir(d, p2).$$

and for that of C',

$$cons(p1, d) \ \wedge \ dir(p1, d) \ \wedge \ dir(d, e) \ \wedge cons(e, p2) \ \wedge dir(e, p2).$$

We need the following relations among $p1$, $p2$, e and d to be preserved in Candidate executions of C'

$$p1 \xrightarrow[hb]{} e \ \wedge \ p1 \xrightarrow[hb]{} d \ \wedge \ d \xrightarrow[hb]{} p2 \ \wedge \ e \xrightarrow[hb]{} p2.$$

After reordering, we do have these relations preserved due to transitivity

$$p1 \xrightarrow[hb]{} d \ \wedge \ d \xrightarrow[hb]{} e \ \Rightarrow \ p1 \xrightarrow[hb]{} e.$$
$$e \xrightarrow[hb]{} p2 \ \wedge \ d \xrightarrow[hb]{} e \ \Rightarrow \ d \xrightarrow[hb]{} p2.$$
$$p1 \xrightarrow[hb]{} d \ \wedge \ d \xrightarrow[hb]{} e \ \wedge \ e \xrightarrow[hb]{} p2 \ \Rightarrow \ p1 \xrightarrow[hb]{} p2.$$

If we can "pivot" the remaining set K_e to $p1$ and K_d to $p2$, it would ensure that our intended relations remain intact after reordering by transitivity. To state formally, we have a valid pair of pivots $< p1, p2 >$ when the following two conditions hold

$$\forall \ k \in K_e - \{p1\}, \ k \xrightarrow[hb]{} p1.$$
$$\forall \ k \in K_d - \{p2\}, \ p2 \xrightarrow[hb]{} k.$$

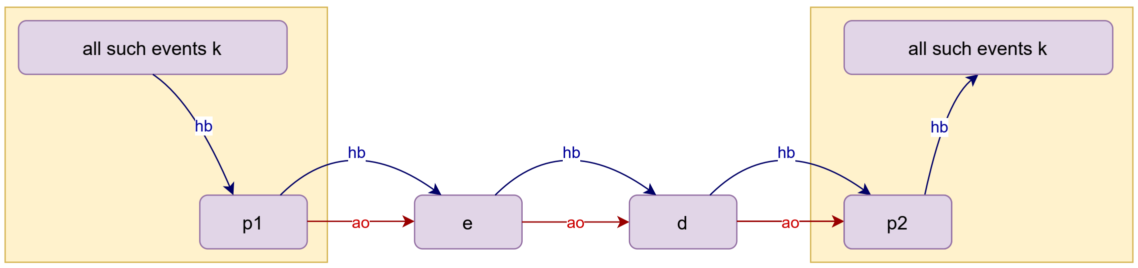


Figure 10: For any Candidate execution, the intuition behind valid pivots $< p1, p2 >$

By lemma 1 and lemma 2 respectively, we have for C, the following condition where $< p1, p2 >$ is a valid pivot pair

$$e\!:\!uo \vee (e\!:\!sc \wedge e \!\in\! W).$$
$$d\!:\!uo \vee (d\!:\!sc \wedge d \!\in\! R).$$

The following table summarizes the cases where we have a valid pair of pivots $< p1, p2 >$

<p1, p2>	R-R	R-W	W-R	W-W
uo-uo	Y	Y	Y	Y
uo-sc	Y	N	Y	N
sc-uo	N	N	Y	Y
sc-sc	N	N	Y	N

Figure 11: Table summarizing whether we have valid pair of pivots based on e and d

We show a simple example where we do not have a valid pair of pivots, particularly because $p1$ is not a valid pivot. Note that in this example, $K_e = K_{e1} + K_{e2} + p1 + p_x$

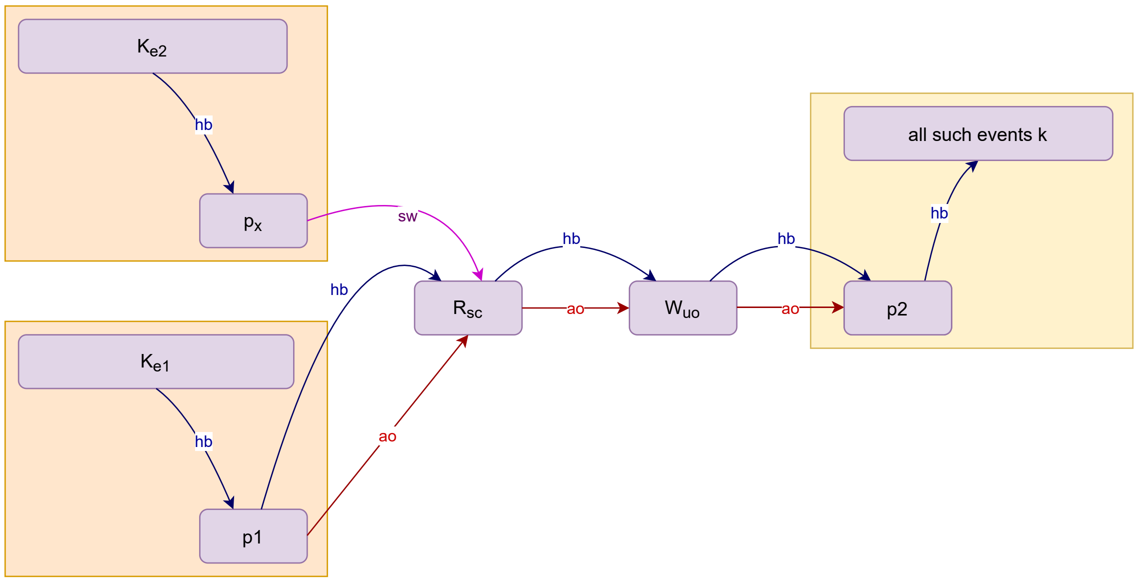


Figure 12: A Candidate Execution where p1 is not a valid pivot

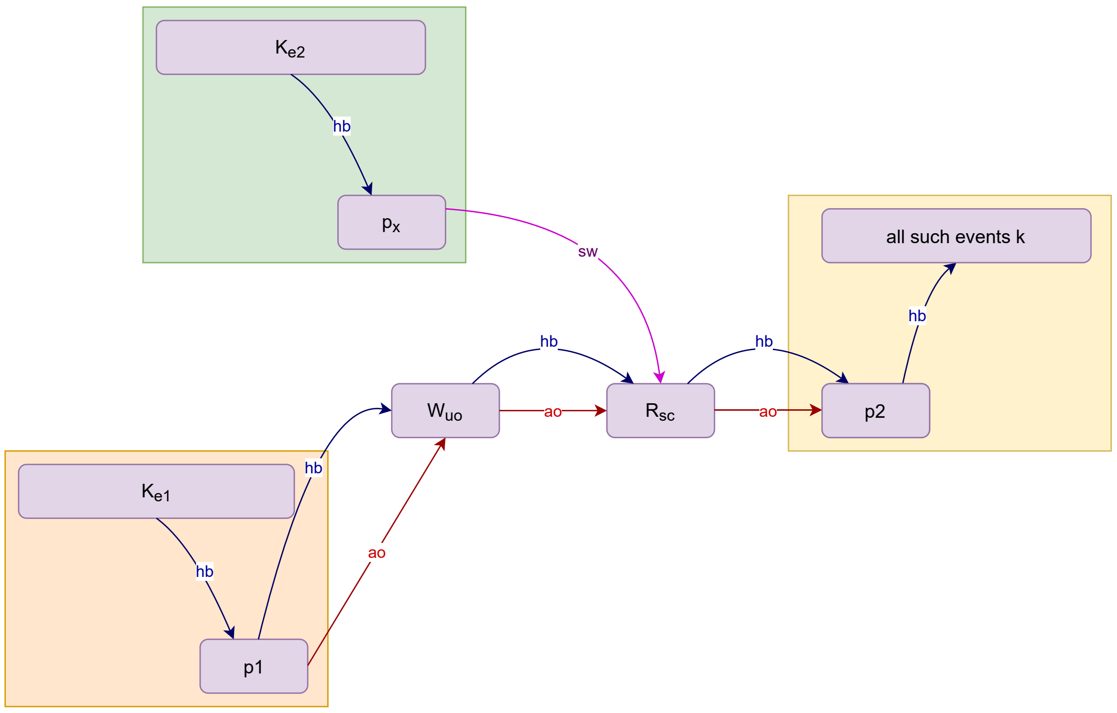


Figure 13: The resultant Candidate Execution after reordering, exposing the relations with p_x, K_{e2} and d that are lost

2. Additional *happens-before* relations Although we have identified the cases when *happens-before* relations are preserved, we also get some additional relations in some of them.

As an example, for the case when d is a sequentially consistent read, by lemma 1, in any execution of C

$$k \xrightarrow[hb]{} d \not\Rightarrow k \xrightarrow[hb]{} e$$

But in *Executions* of candidate C', by transitivity, we have

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

This is because, there are sets of relations that come through certain *synchronize-with* relations. Thus, although we are able to preserve relations that existed in any *CandidateExecution* of C, we also in the process, introduce new ones in *CandidateExecutions* of C'. The figure below shows pictorially an example of a Candidate Execution of C for the case above

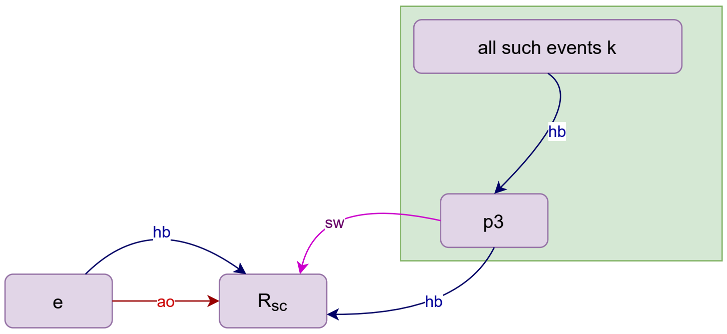


Figure 14: A Candidate Execution where d is a sequentially consistent read

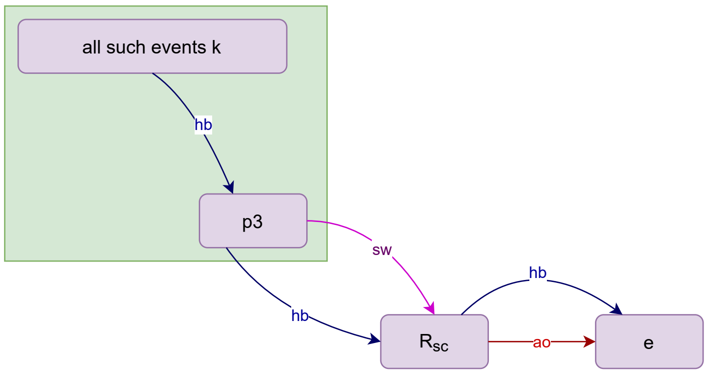
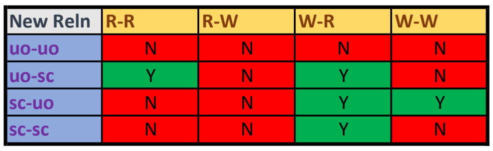


Figure 15: The Candidate Execution after reordering, exposing the new relations established with e, $p3$ and set k

Explain the above figures or perhaps highlight the new relations that are established.

To summarize, the table below shows the cases where new relations could be introduced.



New Reln	R-R	R-W	W-R	W-W
uo-uo	N	N	N	N
uo-sc	Y	N	Y	N
sc-uo	N	N	Y	Y
sc-sc	N	N	Y	N

Figure 16: Table summarizing when new *happens-before* relations could be introduced based on having valid pair of pivots

For these cases, we must know whether these new relations introduce new observable behaviors.

4. Presence of cycles? Before we go into analyzing whether new relations introduce observable behaviours, we first ensure there are no $\xrightarrow[hb]{}$ cycles introduced in the process. Consider the example below

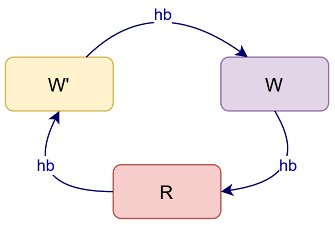


Figure 17: Caption

Notice that here, the axiom of coherent reads restricts R to read from W'.

$$R \xrightarrow[hb]{} W' \Rightarrow \neg R \xrightarrow[rf]{} W'$$

But by transitive property, it is also the case that $W' \xrightarrow[hb]{} R$.

$$W' \xrightarrow[hb]{} W \ \wedge \ W \xrightarrow[hb]{} R \ \Rightarrow \ W' \xrightarrow[hb]{} R$$

As per this, the axiom of coherent reads shouldn't restrict $R \xrightarrow[rf]{} W'$. To avoid such cases, we will need to ensure that no Candidate Execution of C' after e and d are reordered have $\xrightarrow[hb]{}$ cycles.

Note that if a cycle exists after reordering, then

1. The relations preserved do not themselves create a cycle (ref to the theorem)
2. Additional new relations may introduce cycles

The first part is straightforward as we assume we can only do reordering on Candidate Exectuions of C not having cycles.

To address the second part, we first address the cases where $d \xrightarrow[hb]{} e$ may be part of the cycle. The other event k, may be either from the set K_e, K_d or a new relation that is formed.

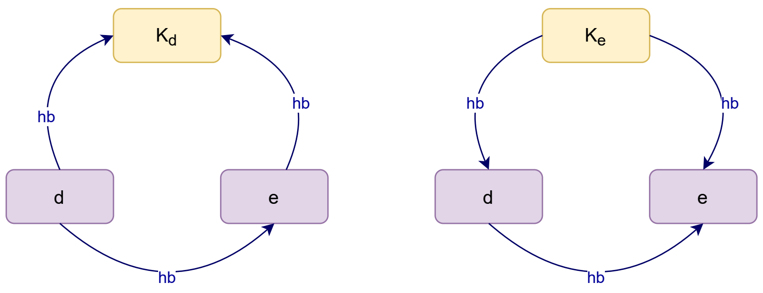


Figure 18: If k belongs to one of the sets K_e or K_d

The above figure shows that k cannot belong to either of the sets, as their relations with e and d will not result in a cycle.

For cases where $k \xrightarrow[hb]{} e$ is the set of new relations, note that by lemma 1

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

For cases where $d \xrightarrow[hb]{} k$ is the set of new relations, by lemma 2

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

So for both these cases also, a cycle with $d \xrightarrow[hb]{} e$ cannot exist. The following figure shows pictorially this fact.

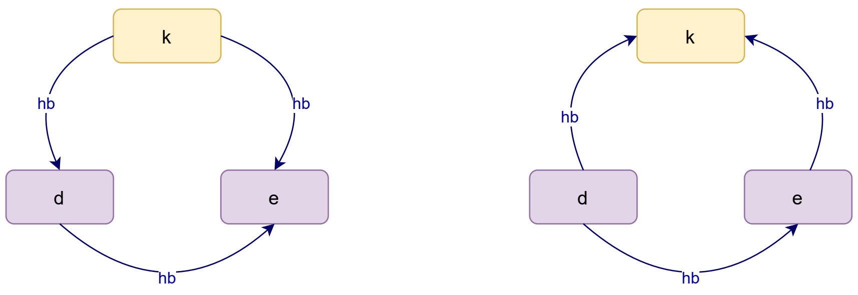


Figure 19: If $k \xrightarrow[hb]{} e$ or $d \xrightarrow[hb]{} k$ are new sets of relations

For the one case where we have two new sets of relations formed, i.e $d \xrightarrow[hb]{} k$ and $k \xrightarrow[hb]{} e$, we could have a case where k is a common event for both sets. But, by lemma 1, we also have $k \xrightarrow[hb]{} d$ and by lemma 2, $e \xrightarrow[hb]{} k$. Thus, we have a cycle. The following figure shows this pictorially

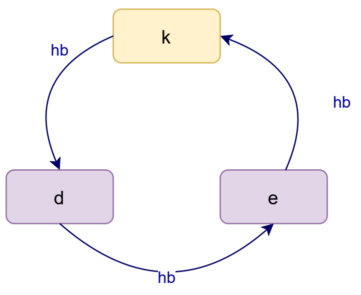


Figure 20: A cycle exists in the case where we have two new sets of relations ($k \xrightarrow[hb]{} e$ and $d \xrightarrow[hb]{} k$)

Maybe have a better figure, meaning a set of relations where each figure shows clearly which relaiton is implied due to whcih lemma

Now for the case when $d \xrightarrow[hb]{} e$ may not be part of the cycle, we have only two other relations, $k \xrightarrow[hb]{} e$ or $d \xrightarrow[hb]{} k$.

Considering the first scenario where the new set of relations are of the form $k \xrightarrow[hb]{} e$. Suppose a cycle exists with another event k'. Then

$$k \xrightarrow[hb]{} e \ \wedge \ e \xrightarrow[hb]{} k' \ \wedge \ k' \xrightarrow[hb]{} k$$

Note that the latter two relations are not new, since the only new set of relations are of the first form. Now, by lemma 1 and by transitivity respectively

$$k \xrightarrow[hb]{} e \Rightarrow k \xrightarrow[hb]{} d$$
$$e \xrightarrow[hb]{} k' \Rightarrow d \xrightarrow[hb]{} k'$$

So, the following is also a cycle

$$k \xrightarrow[hb]{} d \ \wedge \ d \xrightarrow[hb]{} k' \ \wedge \ k' \xrightarrow[hb]{} k$$

But these relations already exist in the original Candidate Execution, which implies a cycle existed before reordering. This contradicts our assumption that we only reorder when the Candidate Executions of C have no cycles. Thus, by contradiction such a cycle cannot exist.

In similar lines for the cases where the set of new relations are of the form $d \xrightarrow[hb]{} k$, we can show by contradiction that a cycle cannot exist.

4. Do new relations introduce new observable behaviors? In any candidate execution, reordering events e and d eliminates the relation $e \xrightarrow[hb]{} d$ and introduces the new relation $d \xrightarrow[hb]{} e$. New behaviours created by the latter directly, if any, are of course intentional (and should normally be avoided by ensuring e and d are independent), but we need to ensure that this does not also result in new behaviours indirectly.

On observing the role on the axioms on this relation, notice that if both e and d are read events then the range does not matter. For all other cases, if events e and d have overlapping ranges, one could introduce a new observable behavior after reordering them (a simple use of Coherent Reads / Sequentially Consistent Atomics).

Any other new relations that are introduced can be divided into 4 cases, in terms of our events e and d and the new relation with some event k:

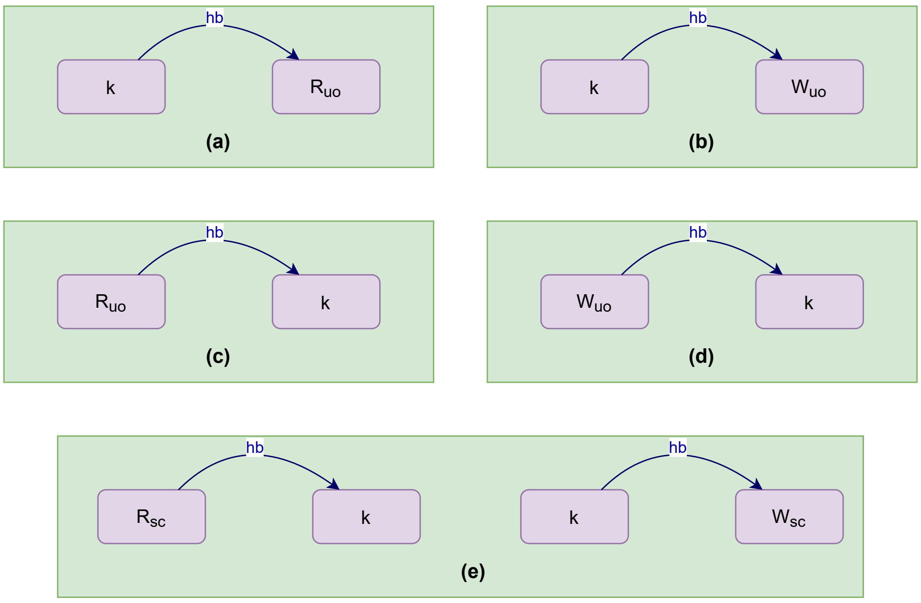


Figure 21: Caption

Change the figure above to represent only the first four cases

In each of the above cases, note firstly that we need to only consider cases where their ranges are overlapping/equal.

Figure below shows a breakdown of sub-cases for the first case (a), varying based on the nature of event k.

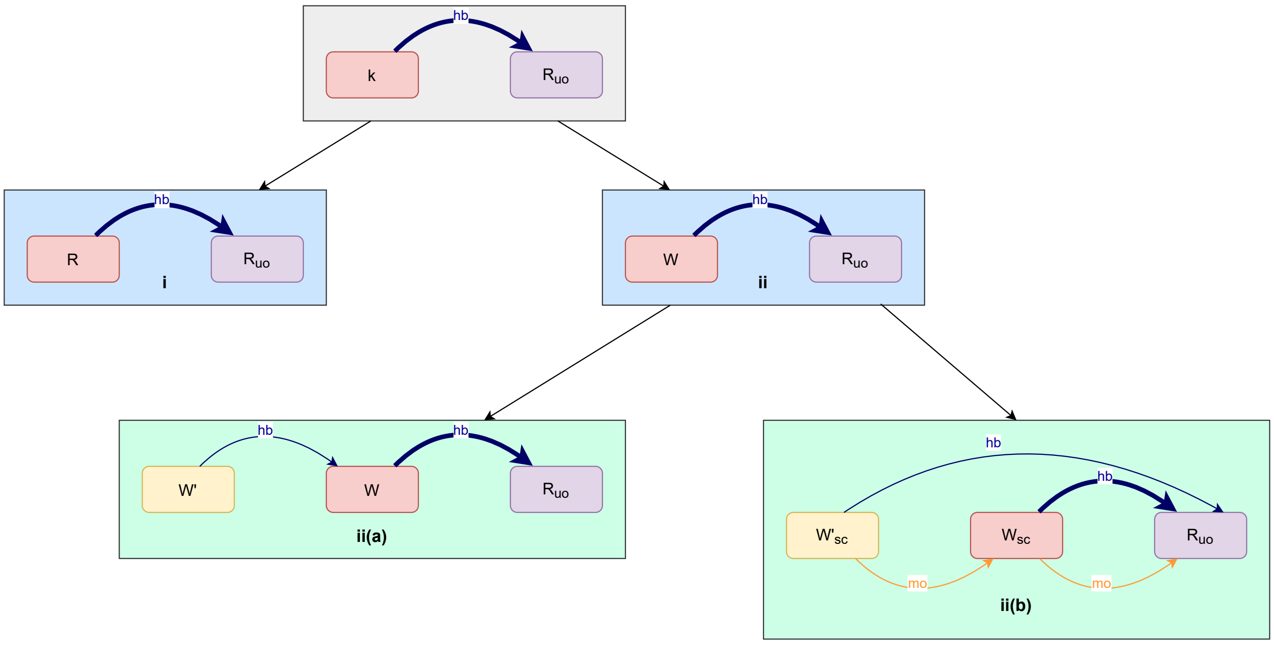


Figure 22: The role of the axioms on introducing a new relation between an unordered Read and some event k

1. For (i), when k is a read, none of the rules have any implications on observable behaviors.
2. For (ii), when k is a write, the rule of coherent reads (ii(a)) or sequentially consistent atomics (ii(b)) could restrict the read (e) from reading overlapping ranges of W' with W.

The above case analysis shows us that the new relation could 'trigger' the consistency rules, only to restrict possible reads-from relations, thus restricting possible observable behaviors.

The other cases, also have instances which can 'trigger' some cases of the axioms, thus restricting possibly some $\overrightarrow{rf}$ relations.b These cases are shown in the figures below:

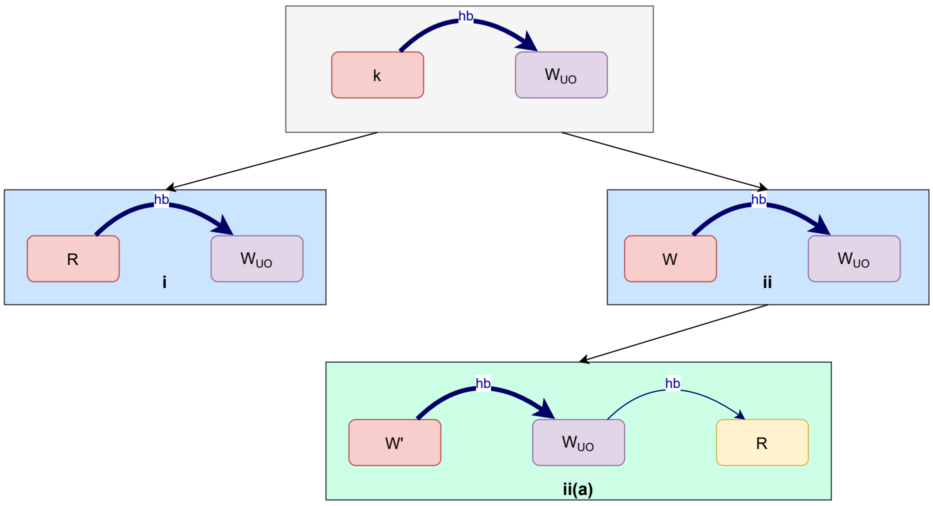


Figure 23: (i) and (ii(b)) satisfy the axiom of Coherent Reads

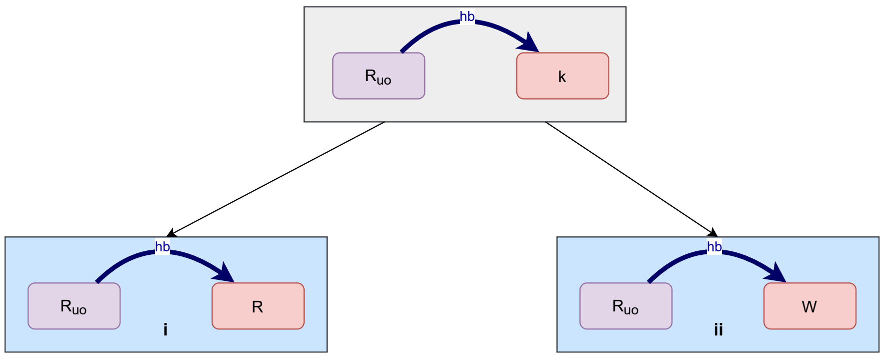


Figure 24: (ii) satisfies the axiom of Coherent Reads

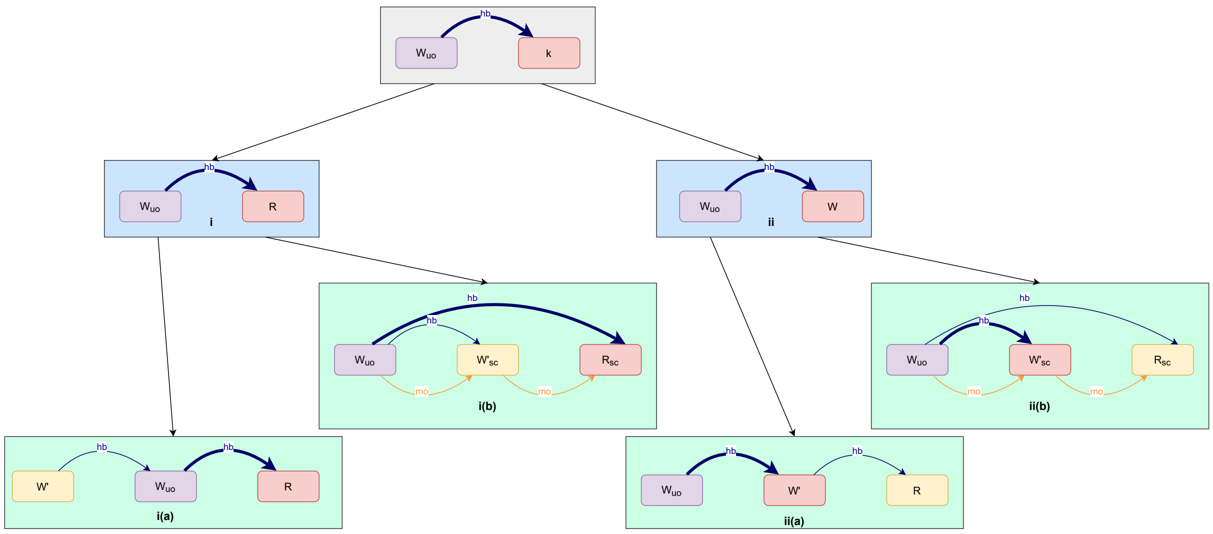
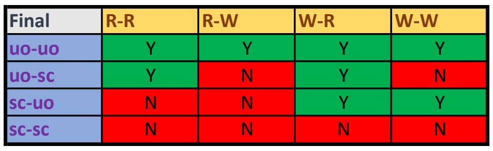


Figure 25: (i(a)), (ii(a)) satisfy the axiom of Coherent Reads, whereas (i(b)), (ii(b)) satisfy the axiom of Sequentially Consistent Atomics

The main reason for this is that we framed he axioms in a form that restricts *reads-from* relations. So in any case where adding an additional *happens-before* relation ”triggers” an axiom, we are bound to have some behaviors restricted. It is this fact that is elicited explicitly by going case wise on all relations that are introduced.

The table below summarizes the valid cases where, we have a pair of valid pivots, where new relations do not introduce new observable behaviors and do not have cycles.



Final	R-R	R-W	W-R	W-W
uo-uo	Y	Y	Y	Y
uo-sc	Y	N	Y	N
sc-uo	N	N	Y	Y
sc-sc	N	N	N	N

Figure 26: The final table summarizing the valid cases where observable behaviors will only be a subset after reordering.

Keep in mind that the comparision of ranges is done while addressing question 3 in the proof, so the table above, implicitly also takes into account only the valid cases where ranges are also correct

The table above, precisely is the definition of a reorderable pair. If we write the above table in the form of an expression we have an expanded format of our Reorderable pair function.

$$
\begin{aligned}
& Reord(e,d) = \\
& (((e\!:\!uo \land d\!:\!uo)\ \land \\
&\qquad ((e\in R \land d\in R)\lor \\
&\qquad\ (e\in W \land d\in R \land (\Re(e)\ \ \&\ \ \Re(d) = \phi))\lor \\
&\qquad\ (e\in R \land d\in W \land (\Re(e)\ \ \&\ \ \Re(d) = \phi))\lor \\
&\qquad\ (e\in W \land d\in W \land (\Re(e)\ \ \&\ \ \Re(d) = \phi)))) \\
&\qquad \lor \\
& ((e\!:\!sc \land d\!:\!uo)\ \land \\
&\qquad ((e\in W \land d\in R \land (\Re(e)\ \ \&\ \ \Re(d) = \phi))\lor \\
&\qquad\ (e\in W \land d\in W \land (\Re(e)\ \ \&\ \ \Re(d) = \phi)))) \\
&\qquad \lor \\
& ((e\!:\!uo \land d\!:\!sc)\ \land \\
&\qquad ((e\in R \land d\in R\land)\lor \\
&\qquad\ (e\in W \land d\in R \land (\Re(e)\ \ \&\ \ \Re(d) = \phi)))))
\end{aligned}
$$

$\square$ $\square$

Corollary 2.1.1. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider two events e and d such that $\neg cons(e,d)$ is true in C and e $\overrightarrow{ao}$ d. Consider another Candidate C' resulting after reordering e and d in C. Then, the set of Observable behaviors possible in C' is a subset of C only if $Reord(e,d)$ and the following holds true.*

$$\forall\ k\ s.t.\ e \underset{ao}{\longrightarrow} k\ \land\ k \underset{ao}{\longrightarrow} d\ .\ Reord(e,k)\ \land\ Reord(k,d)$$

Proof. $\square$

7.4 Counter examples for the invalid cases

7.5