ECMAScript Axiomatic Memory Consistency Model

Akshay Gopalakrishnan

November 2019

0.1 Agents, Events and their Types

0.1.1 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, 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.

Perhaps draw a figure here to represent the role of agent clusters and different shared memory fragments.

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 (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 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 indexis 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 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 $\Re(e) = \Re(d)$

Note that two ranges being overlapping is different from them being equal. This distinction is used to definecertain 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 do not nee Value function as we do not use it anywhere nor is it specified in the standard

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* 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*. 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: $event_{type}$.

Perhaps put the below information on atomic as a footnote? 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.

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.

Perhaps place the blue text below as a footnote.

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.

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 ($\overrightarrow{rbf}$) 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 $\overrightarrow{rbf}$ 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 ($\overrightarrow{rf}$) 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 $e \xrightarrow[rf]{} d2$.

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.

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

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.

0.4 Ordering Relations among Events

Agent Order ($\overrightarrow{ao}$) 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.

Synchronize-With Order ($\overrightarrow{sw}$) 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.

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

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

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

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.)

Happens Before Order ($\overrightarrow{hb}$) A transitive order on events, composed of the following:

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

$$(e \overrightarrow{ao} d) \ \Rightarrow\ (e \overrightarrow{hb} d)$$

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

$$(e \overrightarrow{sw} d) \ \Rightarrow\ (e \overrightarrow{hb} d)$$

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

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

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

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

$$e \overrightarrow{hb} d \Rightarrow e \overrightarrow{mo} d$$

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 $\overrightarrow{ao}$ 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* **??**.

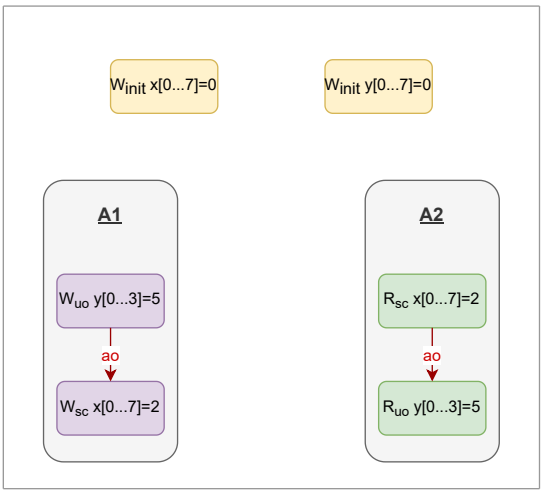

Figure 1: An example of a Candidate

Definition 3. *Candidate Execution A Candidate with the addition of* $\overrightarrow{sw}$, $\overrightarrow{hb}$ *and* $\overrightarrow{mo}$ *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.*

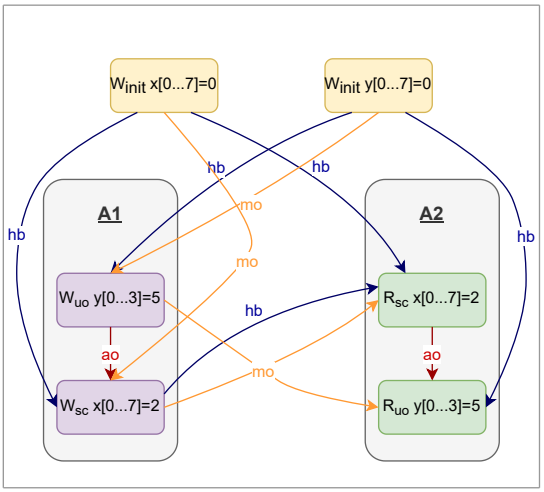

Figure 2: An example of an Execution based on Candidate above

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

Definition 4. *Observable Behavior*

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

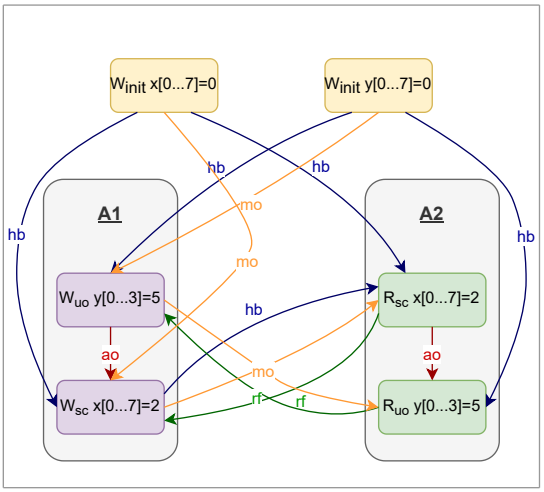

Figure 3: Observable Behavior

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 $\overrightarrow{rf}$ 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 $\overrightarrow{rf}$ relation while having a $\overrightarrow{sw}$ relation among two events.

1 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 $\overrightarrow{hb}$ 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

$$e \overrightarrow{hb} d \ \Rightarrow \ \neg \, e \overrightarrow{rf} d.$$

- 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.

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

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

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

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.

2 Race

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,

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

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 $\boldsymbol{DR}$ 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).*

3 Consistent Executions (Valid Observables)

A valid observable behaviour is when:

1. No $\overrightarrow{rf}$ relation violates the above memory consistency rules.
2. $\overrightarrow{hb}$ 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.

4 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 $\overrightarrow{hb}$ 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 $\overrightarrow{rf}$ (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.

The following definitions and lemmas are not particular to instruction reordering, so I think we can make it a point to put this in a section that introduces our work on optimizations.

4.1 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* $\overrightarrow{ao}$ *relation among them and are* next to each other*, which can be defined formally as*

$$(e \overrightarrow{ao} d \ \wedge \ \nexists k \text{ s.t. } e \overrightarrow{ao} k \ \wedge \ k \overrightarrow{ao} d) \ \vee \ (d \overrightarrow{ao} e \ \wedge \ \nexists k \text{ s.t. } d \overrightarrow{ao} k \ \wedge \ k \overrightarrow{ao} e)$$

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

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

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

4.2 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

$$cons(e,d) \ \wedge \ e \overrightarrow{ao} d \ \wedge \ ((d{:}\,uo) \ \vee \ (d{:}\,sc \ \wedge \ d{\in}W))$$

then,

$$k \overrightarrow{hb} d \Longrightarrow k \overrightarrow{hb} e$$

When we have two consecutive events e and d which are one after the other (i.e. $e \overrightarrow{ao} d$*), we can use transitive property of* $\overrightarrow{hb}$ *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 :

$$cons(e,d) \ \wedge \ e \overrightarrow{ao} d \tag{0}$$

In the first case,

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

Then for any event k

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

An event that satisfies the above with d is e.

$$k = e \qquad from \quad (\textbf{??}, \textbf{??}) \tag{3}$$

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

$$k \overrightarrow{hb} d \Rightarrow k \overrightarrow{hb} d \qquad from \quad (\textbf{??}, \textbf{??}, \textbf{??}, \textbf{??})$$

The following figure should explain this intuition:

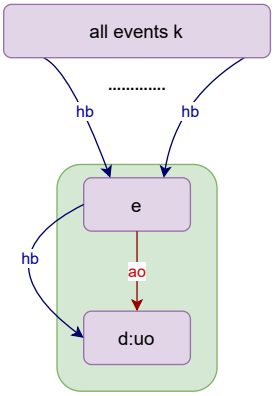

Figure 4: For the first case

In the second case,

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

Then for any event k

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

We once again have event e satisfying the above

$$k = e \qquad from \quad (\mathbf{??}, \mathbf{??}) \tag{6}$$

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 $k \neq e$

$$k \xrightarrow[hb]{} d \Rightarrow k \xrightarrow[hb]{} d \qquad from \quad (\mathbf{??}, \mathbf{??}, \mathbf{??}, \mathbf{??})$$

The following figure explains this intuition:

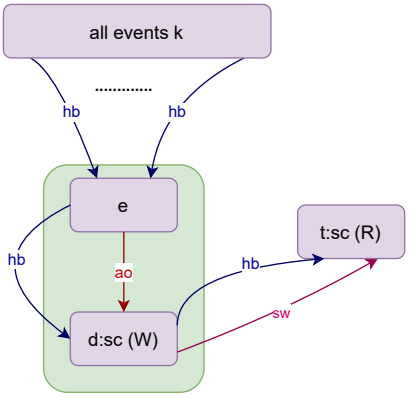

Figure 5: For the second case

□

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

If

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

then,

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

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:

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

In the first case,

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

Then for any event k

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

The event that satisfies the above with e is d

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

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

$$e \xrightarrow[hb]{} k \Rightarrow d \xrightarrow[hb]{} k \qquad from \quad (??, ??, ??, ??)$$

The following figure should explain this intuition:

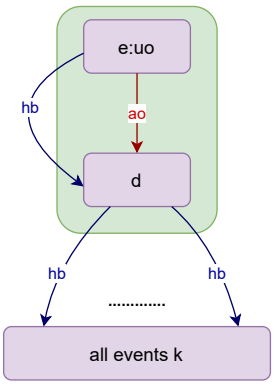

Figure 6: Caption

In the second case,

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

Then for any event k

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

We once again have event d satisfying the above

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

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 $k \neq d$

$$e \xrightarrow[hb]{} k \Rightarrow d \xrightarrow[hb]{} k \qquad from \quad (??, ??, ??, ??)$$

The following figure explains this intuition:

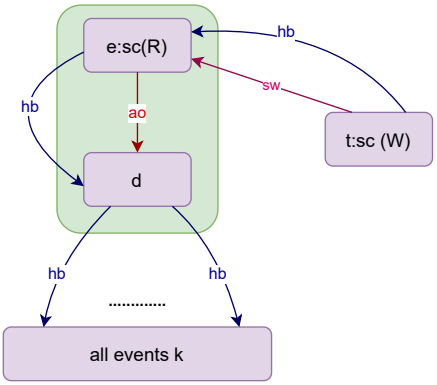

Figure 7: Caption

□

4.3 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.*

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

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 $\overrightarrow{hb}$ relations (except $e \overrightarrow{hb} d$) and that any new $\overrightarrow{hb}$ 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 $e \overrightarrow{hb} d$ 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 $e \overrightarrow{hb} d$, do other *happens-before* relations remain intact?
2. Apart from $d \overrightarrow{hb} e$, 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 $\overrightarrow{hb}$ relations among events are lost after reordering, we may introduce new observable behaviors. The relations that are subject to change can be divided into four parts using events e and d

a) $k \overrightarrow{hb} e$ b) $e \overrightarrow{hb} k$

c) $d \overrightarrow{hb} k$ d) $k \overrightarrow{hb} d$

Firstly, note that the relations of the form $e \overrightarrow{hb} k$ come through either a $\overrightarrow{sw}$ relation with e or relations through event d, i.e. of the form $d \overrightarrow{hb} k$. The ones that come due to the latter, may not be preserved after reordering, if we strictly are only able to derive them with relations through d. Note also that, a similar argument exists for relations of the form $k \overrightarrow{hb} d$ wherein relations derived through e ($k \overrightarrow{hb} e$) may be lost after reordering.

Hence, 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.

$$K_e = \{k \mid k \overrightarrow{hb} e\}.$$
$$K_d = \{k \mid d \overrightarrow{hb} k\}.$$

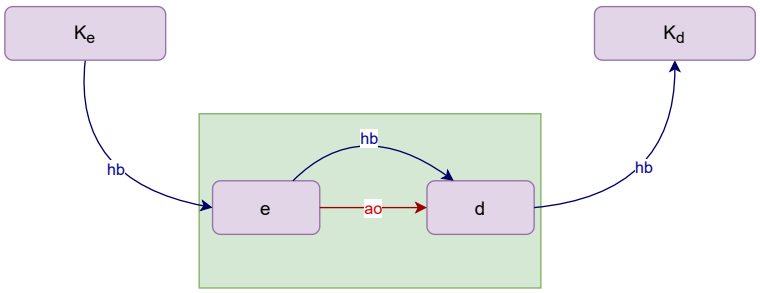

Figure 8: For any Candidate Execution of C, the set K_e and K_d

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 dummyevents 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 followingchanges

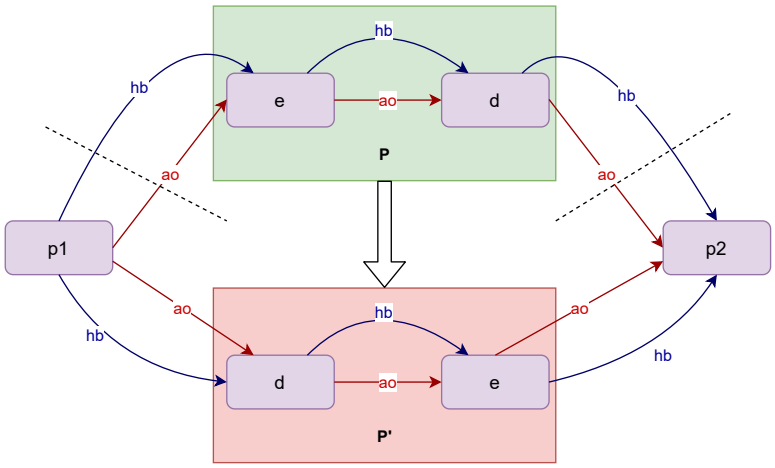

Figure 9: The direct relation changes that can be observed while reordering events e and d

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 key relations to be preserved in Candidate executions of C'

a) $p1 \ \overrightarrow{hb} \ e$

b) $e \ \overrightarrow{hb} \ k$

c) $d \ \overrightarrow{hb} \ p2$

d) $k \ \overrightarrow{hb} \ d$

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

$$p1 \ \overrightarrow{hb} \ d \ \wedge \ d \ \overrightarrow{hb} \ e \ \Rightarrow \ p1 \ \overrightarrow{hb} \ e.$$
$$e \ \overrightarrow{hb} \ p2 \ \wedge \ d \ \overrightarrow{hb} \ e \ \Rightarrow \ d \ \overrightarrow{hb} \ p2.$$
$$p1 \ \overrightarrow{hb} \ d \ \wedge \ d \ \overrightarrow{hb} \ e \ \wedge \ e \ \overrightarrow{hb} \ p2 \ \Rightarrow \ p1 \ \overrightarrow{hb} \ p2.$$

The other two forms of relations may not be preserved due to $d \ \overrightarrow{sw} \ k$ or $k \ \overrightarrow{sw} \ d$. If we can "pivot" the set K_e to $p1$ and K_d to $p2$, it would ensure that our other two intended relations also remain preserved after reordering by transitivity. To state formally, we have a valid pair of pivots $< p1, p2 >$ when the following two conditions hold

$$\forall \ k \in K_e - \{p1\}, \ k \ \overrightarrow{hb} \ p1.$$
$$\forall \ k \in K_d - \{p2\}, \ p2 \ \overrightarrow{hb} \ k.$$

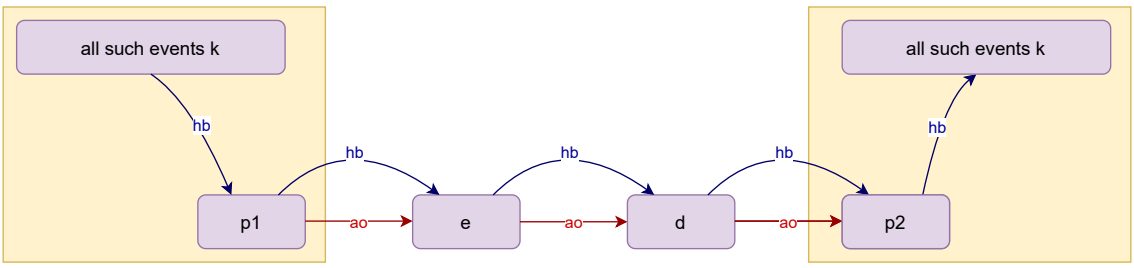

Figure 10: For any Candidate execution, the intuition behind valid pivots $< p1, p2 >$

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 thatin this example, $K_e = K_{e1} + K_{e2} + p1 + p_x$

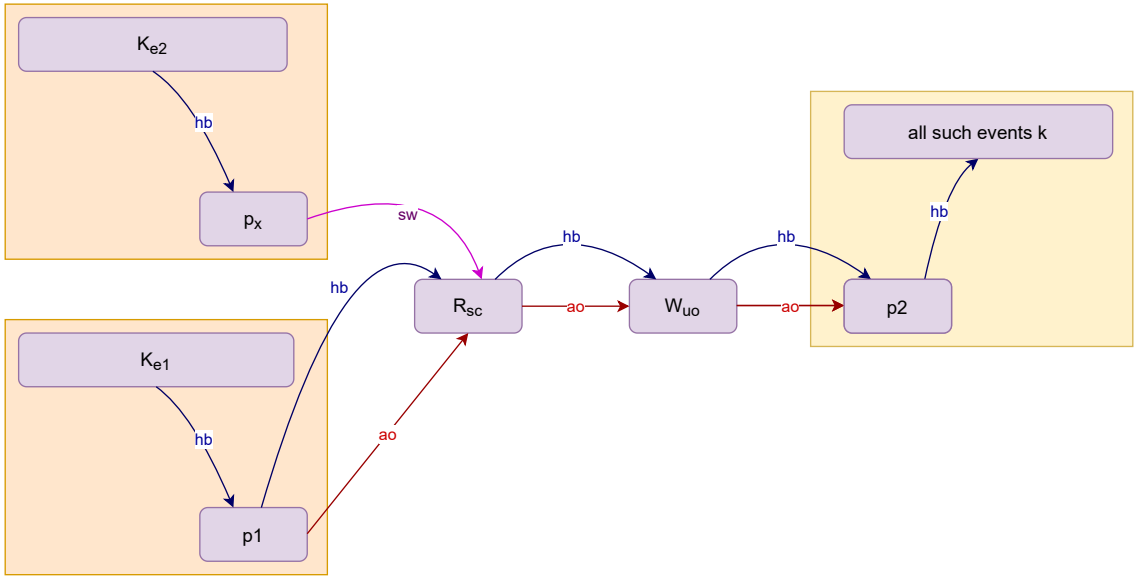

Figure 12: A Candidate Execution where p1 is not a valid pivot

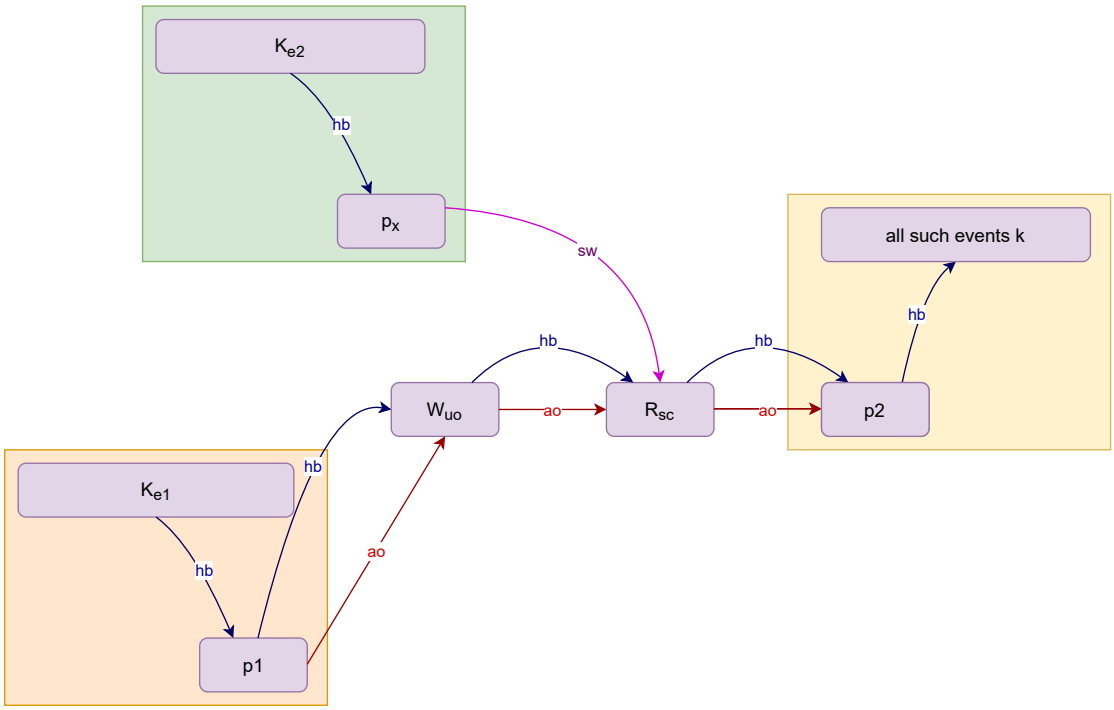

Figure 13: The resultant Candidate Execution after reordering, exposing the relations with p_x, K_{e2} and d that arelost

Strictly speaking, it is not that the happens-before relations are preserved, but that the properties between different happens before relations hold, which implies that for any possible Candidate Execution after reordering, the set of happens-before relatiosn apart from that between e and d remain the same. I am emphasizing this point because we view reordering as just changing agent order between two events, which just needs information from Candiadate but not all its possible Executions.

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 *synchronize-with* relations that d has. 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

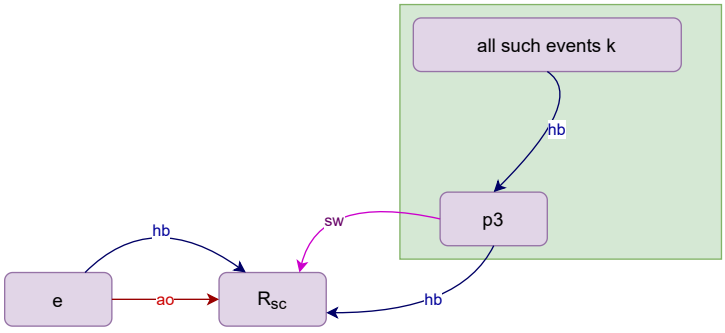

Figure 14: A Candidate Execution where d is a sequentially consistent read

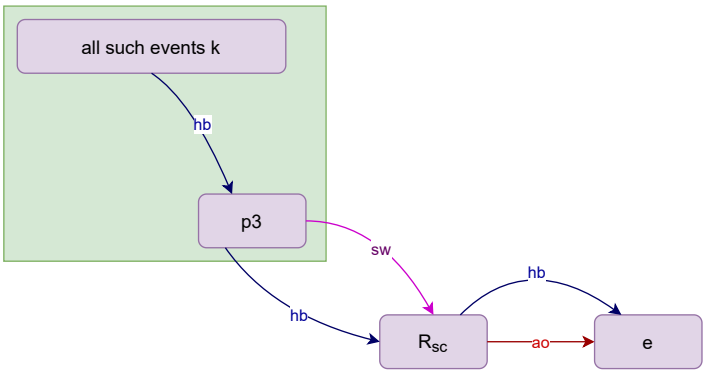

Figure 15: The Candidate Execution after reordering, exposing the new relations established with e, $p3$ and set k

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.

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

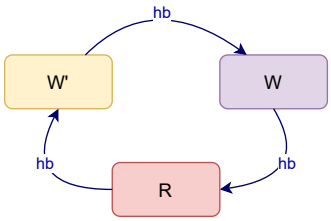

Figure 17: Caption

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

$$R \overrightarrow{hb} W' \Rightarrow \neg R \overrightarrow{rf} W'$$

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

$$W' \overrightarrow{hb} W \ \wedge \ W \overrightarrow{hb} R \ \Rightarrow \ W' \overrightarrow{hb} R$$

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

For 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.

K_e and K_d only apart from the new relation because these are the only valid cases where happens-before relations are preserved after reordering. So we need not consider cases such thta $e \xrightarrow[hb]{} k$ or $k \xrightarrow[hb]{} d$ as the old relationsas they are covered by K_e and K_d.

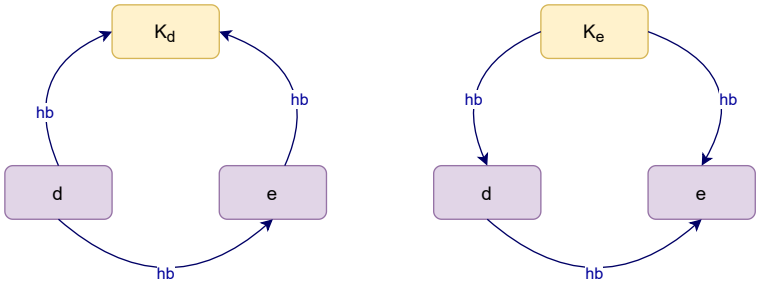

Figure 18: If k belongs to one of the sets K_e or K_d

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.

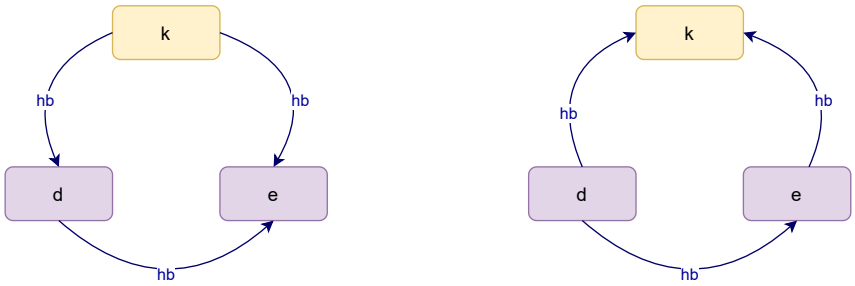

Figure 19: If $k \xrightarrow[hb]{} e$ or $d \xrightarrow[hb]{} k$ are new sets of relations

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

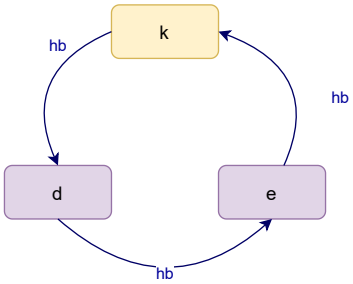

Figure 20: A cycle exists in the case where we have two new sets of relations ($k \xrightarrow[hb]{} e$ and $d \xrightarrow[hb]{} k$)

It is not actually due to lemmas, but just that the new relations were derived through e or d, as these relations existed before reordering.

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 new 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 afterreordering them (a simple use of Coherent Reads / Sequentially Consistent Atomics).

We will later show counter examples for each of the cases that we discard as invalid to reorder.

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

a) $e\!:\!uo \ \wedge \ e\!\in\!R \ \wedge \ k \xrightarrow[hb]{} e.$

b) $e\!:\!uo \ \wedge \ e\!\in\!W \ \wedge \ k \xrightarrow[hb]{} e.$

c) $d\!:\!uo \ \wedge \ d\!\in\!R \ \wedge \ d \xrightarrow[hb]{} k.$

d) $d\!:\!uo \ \wedge \ d\!\in\!W \ \wedge \ d \xrightarrow[hb]{} k.$

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.

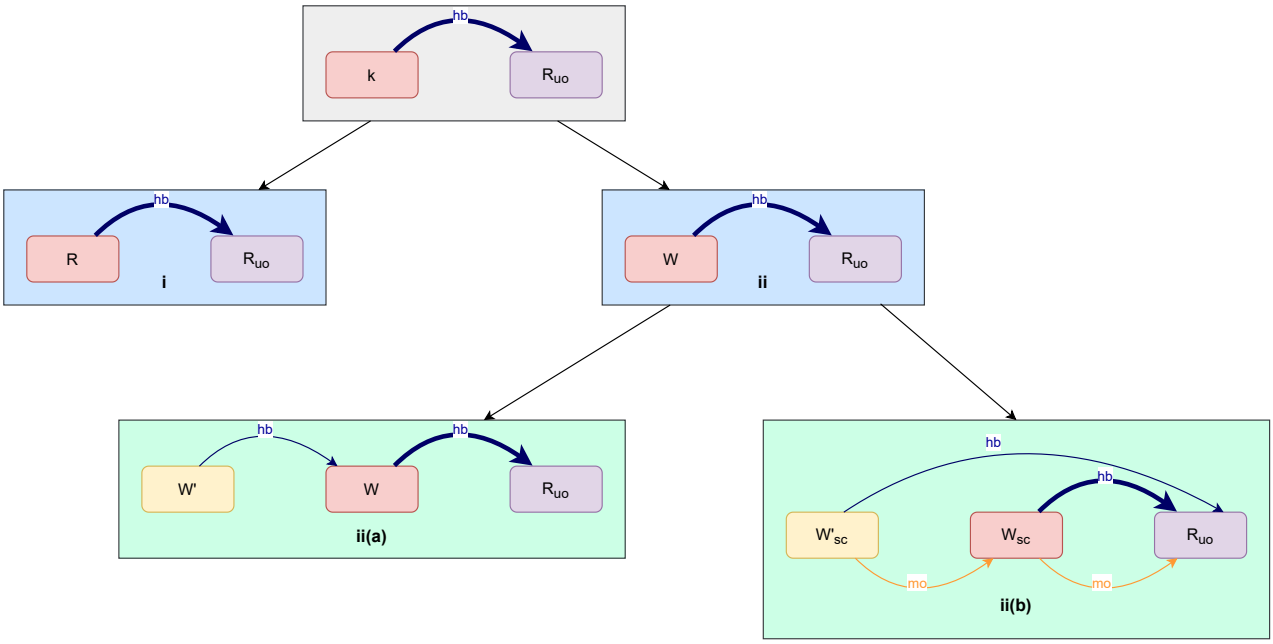

Figure 21: The role of the axioms on introducing a new relation between an unordered Read and some event k

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)) couldrestrict 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. These cases are shown in the figures below:

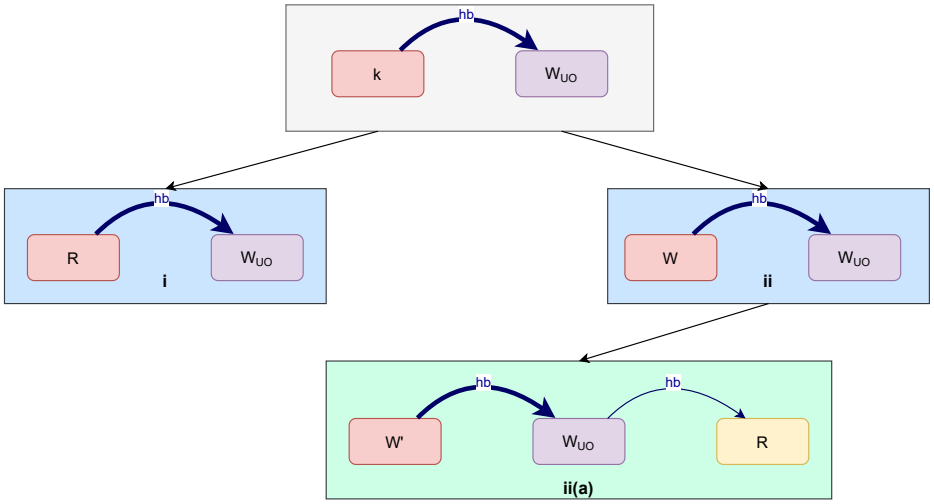

Figure 22: (i) and (ii(b)) satisfy the axiom of Coherent Reads

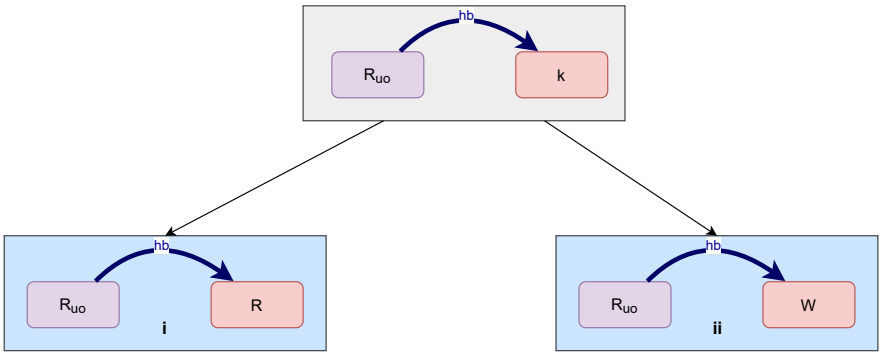

Figure 23: (ii) satisfies the axiom of Coherent Reads

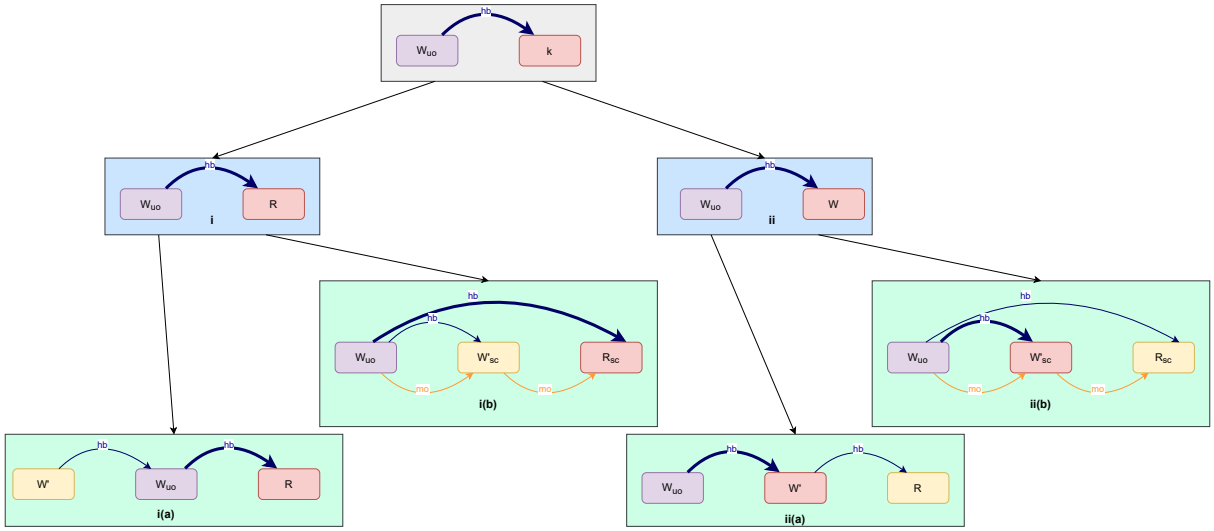

Figure 24: (i(a)), (ii(a)) satisfy the axiom of Coherent Reads, whereas (i(b)), (ii(b)) satisfy the axiom of SequentiallyConsistent Atomics

The main reason for this is that we framed he axioms in a form that restricts *reads-from* relations. Soin any case where adding an additional *happens-before* relation "triggers" an axiom, we are bound to have somebehaviors 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 25: 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.

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

□ □

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* $e \xrightarrow[ao]{} d$*. 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.*

$$\forall\ k\ s.t.\ e \xrightarrow[ao]{} k\ \wedge\ k \xrightarrow[ao]{} d\ .\ Reord(e,k)\ \wedge\ Reord(k,d)$$

Proof. We prove this by induction of number of events k between e and d. Let n denote the number of events.

Base Case: $n = 1$. This means we have one event k such that

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

What we want after reordering is

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

Without loss of generality, we can choose to first reorder k and d. For this we have $cons(k,d)$ to be true. To reorder, what we only need is $Reord(k,d)$ to hold. Thus, after reordering, we have

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

Similarly, now we need $Reord(e,d)$ to hold to reorder them above, after doing so, we will get

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

Now lastly, we need to reorder e and k for which we need $Reord(e,k)$ to hold, thus giving us our final result

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

The conditions required to do the reordering is in line with the requirement stated by the corollary.

2. Inductive Case $n > 1$ Assume the above corollary holds for $n = t$.

We need to show that for $n = t + 1$, the corollary still holds, for this note firstly that, we have the following ordering relations.

$$e \xrightarrow[ao]{} k_1 \xrightarrow[ao]{} k_2 \xrightarrow[ao]{} k_3 \xrightarrow[ao]{} ... \xrightarrow[ao]{} k_t \xrightarrow[ao]{} k_{t+1} \xrightarrow[ao]{} d$$

Without loss of generality, we can first reorder k_{t+1} and d. To do this, we need $Reord(k_{t+1}, d)$ to hold, thus giving us the resultant ordering.

$$e \xrightarrow[ao]{} k_1 \xrightarrow[ao]{} k_2 \xrightarrow[ao]{} k_3 \xrightarrow[ao]{} ... \xrightarrow[ao]{} k_t \xrightarrow[ao]{} d \xrightarrow[ao]{} k_{t+1}$$

Now we have t such events between e and d. With our assumption, we can reorder e and d, thus giving us

$$d \xrightarrow[ao]{} k_1 \xrightarrow[ao]{} k_2 \xrightarrow[ao]{} k_3 \xrightarrow[ao]{} ... \xrightarrow[ao]{} k_t \xrightarrow[ao]{} e \xrightarrow[ao]{} k_{t+1}$$

Finally, we need to reorder e and k_{t+1} to get our final result, for which we need $Reord(e, k_{t+1})$ to hold, thus giving us finally

$$d \xrightarrow[ao]{} k_1 \xrightarrow[ao]{} k_2 \xrightarrow[ao]{} k_3 \xrightarrow[ao]{} ... \xrightarrow[ao]{} k_t \xrightarrow[ao]{} k_{t+1} \xrightarrow[ao]{} e$$

Thus we can see that we need two more conditions to hold for us to ensure we can reorder e amd d with $n = t + 1$. This is exactly what the corollary requires.

Hence, by induction the proof is complete.

Observe that wherever our argument states the requirement of *Reord* to hold between two events, it is also the case that those two events are consecutive and have an agent ordering exactly as our Theorem states.

□

4.4 Counter examples for the invalid cases

For each case where reordering is not safe to do, we also show counter examples of programs where new observable behaviors are introduced. This additionally potrays additional proof of the validity of our approach.

For all the examples we show here, we only show the ordering relations that are important to observe. Putting all the relations among different events in the example will result in confusion, hence we avoid doing so.

Reads to same memory where e is of type sc while d is of either uo/sc The following example involves two reads to the same memory and a write.

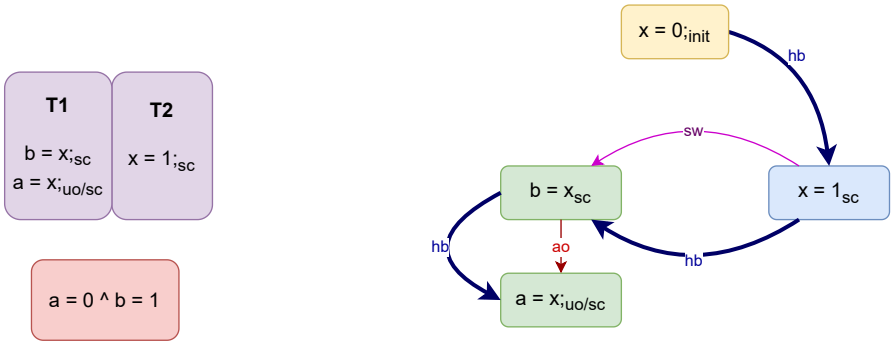

Figure 26: Case where a = 0 , b = 1 is invalid due to Coherent Reads

The figure on the left above shows an example of a candidate where the case of reads in the red box is not possible. The figure on the right shows the Candidate Execution of such a case. Observations:

- We can infer from the Candidate Execution that $\{x = 0_{init}\} \xrightarrow[hb]{} \{x = 1_{sc}\} \xrightarrow[hb]{} \{a = x_{uo/sc}\}$.
- By the second axiom of coherent reads, it is not possible for a to read the value of 0 as x due to the intervening write whiCh changes x to 1.
- This inference does not rely upon the access mode of the read a.

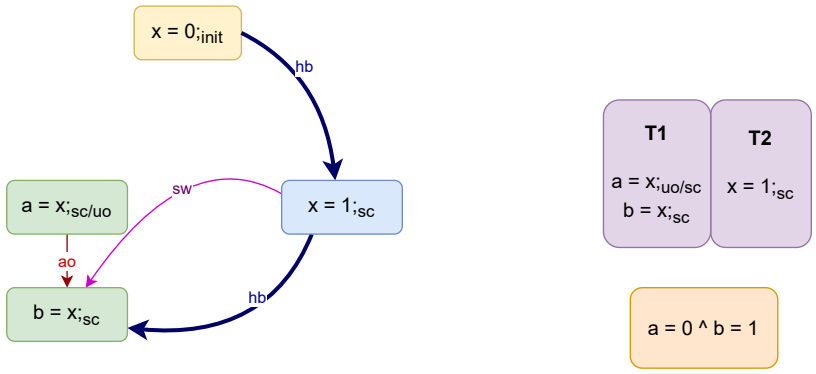

Figure 27: Case where the reads are reordered and a = 0 , b = 1 is valid

The figure on the right shows the program after reordering the two reads in $T1$, where the case of reads in the orange box is possible. The figure on the left shows the Candidate Execution of such a case.
Observations:

- From the Candidate Execution, we can infer $\neg\{x = 0_{init}\} \xrightarrow{hb} \{x = 1_{sc}\} \xrightarrow{hb} \{a = x_{uo/sc}\}$
- We can also infer that $\{x = 0_{init}\} \xrightarrow{hb} \{a = x_{uo/sc}\}$
- Since none of the axioms disallow the above pattern, a is allowed to read the value of x to be 0.
- Hence, the reordering of the two reads is invalid.

Reads to non-equal range of memory where e is of type sc while d is of either uo/sc

A Read e of type sc followed by a Write of either uo/sc The following is an example of a program with a sequentially consistent read followed by a write of any type.

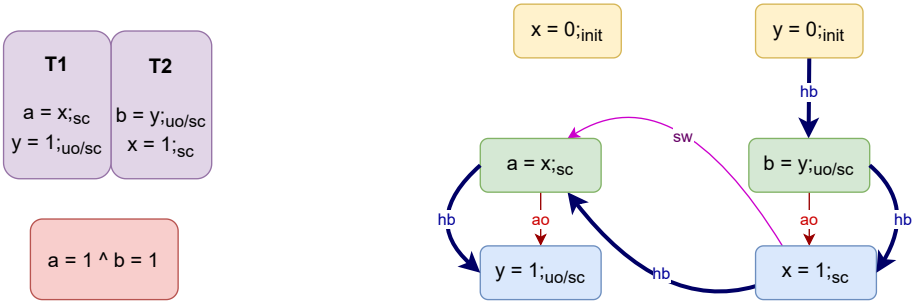

Figure 28: Case where a = 1 and b = 1 is invalid due to Coherent Reads.

The figure on the left above shows an example of a candidate where the case of reads in the red box is not possible. The figure on the right shows the Candidate Execution of such a case. Observations:

- From the Candidate Execution, we can infer $b = y_{uo/sc} \xrightarrow{hb} y = 1_{uo/sc}$
- By the first rule of coherent reads, b cannot read the value of 1 as y.
- This inference was due to $x = 1_{sc} \xrightarrow{hb} a = xsc$

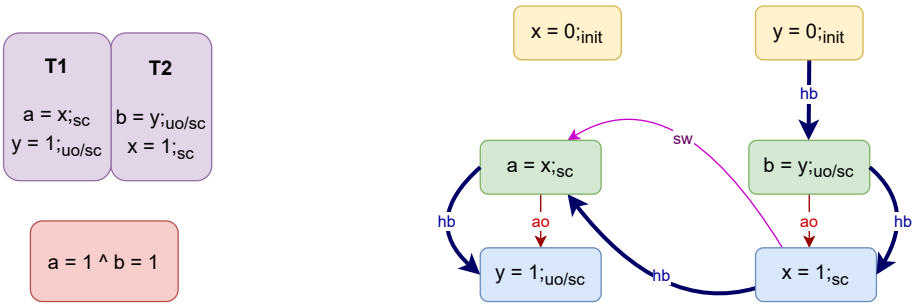

Figure 29: Case where events of T1 are reordered, resulting in a = 1 and b = 1 to be valid.

The figure on the right above shows the program after reordering the two events in $T1$ where case of reads in the orange box is possible. The figure on the left shows the Candidate Execution of such a case. Observations:

- From the Candidate Execution, we can infer $\neg b = y_{uo/sc} \xrightarrow{hb} y = 1_{uo/sc}$
- Since there is no $\xrightarrow{hb}$ relation among the above two events, b can read the value of y as 1.

A Read e of type uo followed by a write d of type sc For this we can use the same example for the previous part (tag figure of example), where we just reorder $T2$'s events.

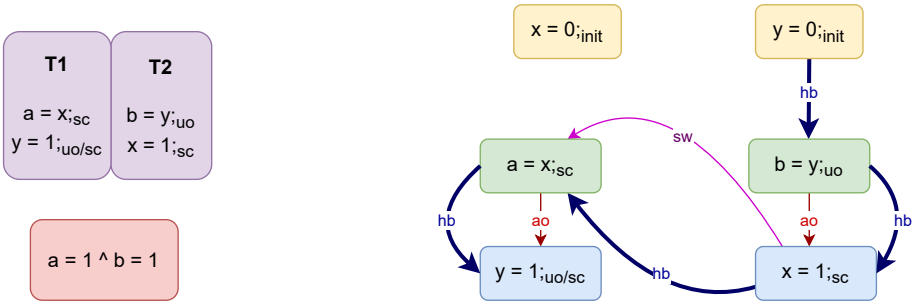

Figure 30: Case where a = 1 and b = 1 is invalid due to Coherent Reads.

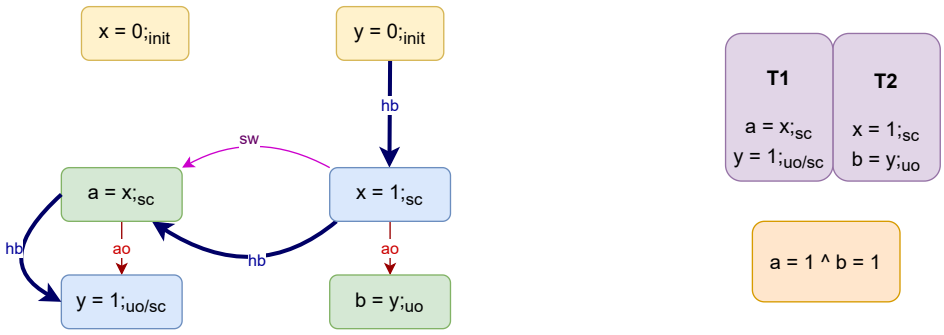

Figure 31: Case where events of T2 are reordered, resulting in a = 1 and b = 1 to be valid.

A Write e followed by a Read d both of type sc A counter example for this is different. It is not the Observable Behavior we are concerned with that is introduced, but that which is allowed but creates a $\xrightarrow{hb}$ cycle. The following example is as such:

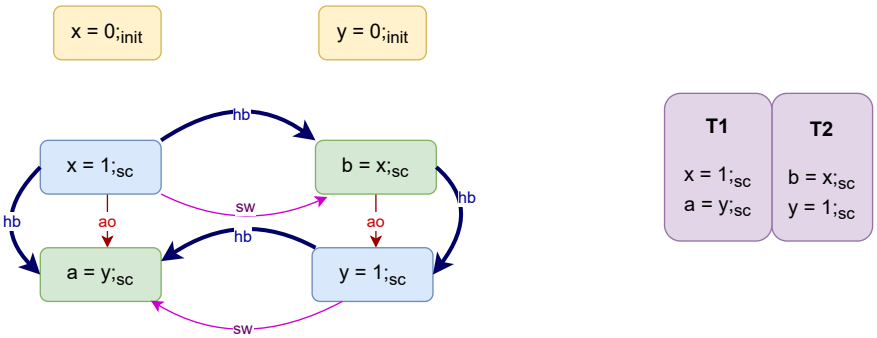

Figure 32: Case where a = 1 and b = 1 is valid and no happens-before cycles

After reordering the two events of $T1$ in the above example, the same observable behavior holds, but has a cycle introduced. One might think that simply discarding that execution would do. But this would mean discarding $\overrightarrow{hb}$ relations also, which would require more information to infer which relations are going to create such cycles and which are not. Since we place no assumptions on these relations, but that any happens-before relation other than the one we remove explicitly be reordering are all possible. Hence, the following reordered program outcome is something we do not risk to allow.

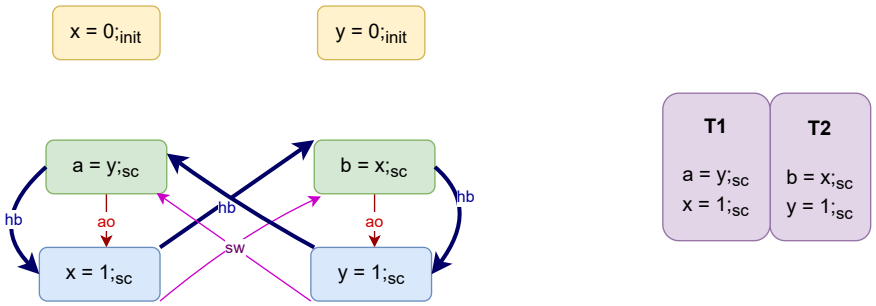

Figure 33: Case where a = 1 and b = 1 is creates a happens-before cycle

Observation:

- From the read values we can infer that the Candidate Execution should have $x = 1_{sc} \overrightarrow{hb} a = x_{sc}$ and $y = 1_{sc} \overrightarrow{hb} a = y_{sc}$.
- The above relations create the cycle $a = y_{sc} \overrightarrow{hb} x = 1_{sc} \overrightarrow{hb} a = x_{sc} \overrightarrow{hb} y = 1_{sc} \overrightarrow{hb} a = y_{sc}$.
- This execution is invalid.

A Write e of type uo/sc followed by a Write d of type sc The following example shows a program with a thread having a write of any access mode(uo/sc) followed by a write of type sc.

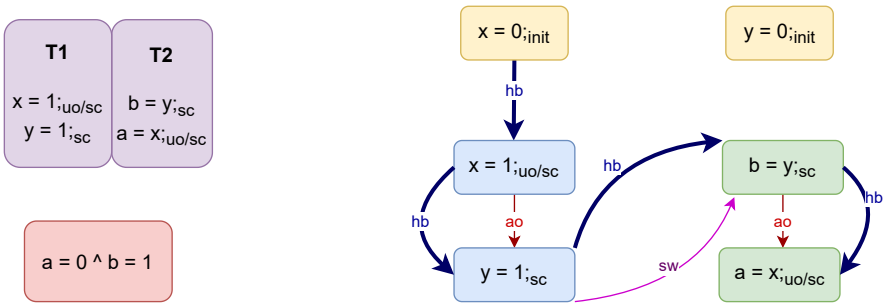

Figure 34: Case where a = 0 and b = 1 is invalid due to Coherent Reads.

The figure on the left above shows an example of a candidate where the case of reads in the red box is not possible. The figure on the right shows the Candidate Execution of such a case. Observations:

- From the Candidate Execution, we can infer $x = 0_{init} \xrightarrow[hb]{} x = 1_{uo/sc} \xrightarrow[hb]{} a = x_{uo/sc}$
- This is a pattern that matches the second axiom of Coherent reads, thus restricting the read of a to have the value of 0 as x.
- This inference was due to $y = 1_{sc} \xrightarrow[hb]{} b = y_{sc}$.

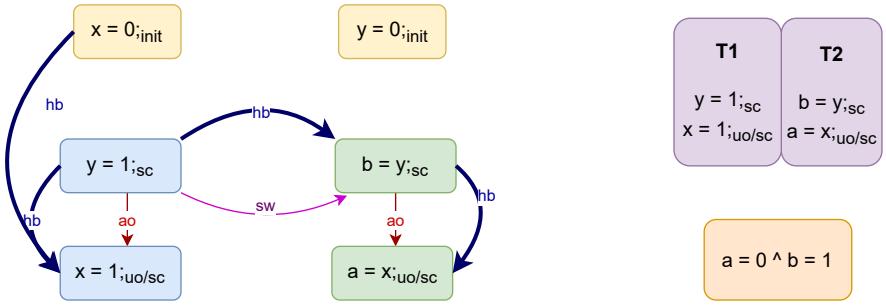

Figure 35: Case where events of T1 are reordered, resulting in a = 0 and b = 1 to be valid.

The figure on the right above shows the program after reordering the two events in $T1$ where case of reads in the orange box is possible. The figure on the left shows the Candidate Execution that explains the orange box case. Observations:

- From the Candidate Execution, we can infer $\neg x = 1_{uo/sc} \xrightarrow[hb]{} a = x_{uo/sc}$
- Hence, there is no pattern that the axioms restrict, thus validating x to be read as 0 by a.

All the above counter examples have a lot of repetitive text and can have better formal arguments than a list of observations. Make plans to clean up this once filled in with all the counter examples.

4.5 From Candidates to Program

To investigate the validity of reordering at the program level, we first, in terms of candidates, address code motion. The following two corollaries cover them:

Corollary 2.1.2. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider a set of events $k_{i\in[1,n]}$ such that $k_i \xrightarrow[ao]{} k_{i+1} \wedge cons(k_i, k_{i+1})$. Consider an event e such that*

$$cons(e, k_1) \ \wedge \ e \xrightarrow[ao]{} k_1 \tag{1}$$

Consider another candidate C' with the only differnence from C being $cons(e, k_n) \ \wedge \ k_n \xrightarrow[ao]{} e$. Then the set of observable behaviors of C' is a subset of that of C if

$$\forall i \in [1, n] \ . \ Reord(e, k_i) \tag{2}$$

Proof. Apply theorem of reordering successively, and by transititvity of subset relations, the corollary holds. □

Corollary 2.1.3. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider a set of events $k_{i\in[1,n]}$ such that $k_i \xrightarrow[ao]{} k_{i+1} \wedge cons(k_i, k_{i+1})$. Consider an event d such that*

$$cons(d, k_n) \ \wedge \ k_n \xrightarrow[ao]{} d \tag{3}$$

Consider another candidate C' with the only differnence from C being $cons(d, k_1) \ \wedge \ d \xrightarrow[ao]{} k_1$. Then the set of observable behaviors of C' is a subset of that of C if

$$\forall i \in [1, n] \ . \ Reord(k_i, d) \tag{4}$$

Proof. Apply theorem of reordering successively, and by transititvity of subset relations, the corollary holds. □

The above corollaries defined is the general form of reordering, which also defines code motion. It is interesting to note that we first noted reordering as reversing agent order, but we never considered it as strictly as we should have. Perhaps make the corollary for non-consecutive reordering of two events more precise.

We first consider programs with conditionals. The following property holds for any candidates of programs having conditional branching.

Property 1. *Candidates of Programs with Conditionals (2-branch) Let $B1, B2$ be two sets of events based on each branch of a conditional in a program P. Let C be any Candidate of P, Consider $b1, b2$ to be representative of any event in $B1, B2$ respectively. Then:*

$$\nexists C \in P \ s.t. \ b1 \in C \ \wedge \ b2 \in C$$

There cannot exist any candidate of the program such that events from both sets can be part of it.

Property 2. *Candidates of Programs with Conditionals (1-branch) Let $B1$ be two sets of events based on each branch of a conditional in a program P. Let C be any Candidate of P, Consider $b1$ to be representative of any event in $B1$. Then:*

$$\exists C \in P \ s.t. \ b1 \notin C$$

There exists a candidate of the program such that events from the branch cannot be part of it.

While the property for 1 branch may not always hold (it can be the case that the branch is always taken in any execution) we are defining it for any program.

Proof. Based on an exeuction of the program, the conditional will either be satisfied or not, but never both. Hence proved both properties. Do we need an elaborate proof of this? As this is direct from existing literature on sequential programs. □

Perhaps we need a general corollary for program level

Corollary 2.1.4. *Reordering under Program with Conditionals Consider a program P and its candidates $C_1, C_2, ..., C_n$ in which events e and d present in all of them with $e \overrightarrow{ao} d$. Consider the set of corresponding candidates $C'_1, C'_2, ..., C'_n$ after reordering e and d and its corresponding program P'. Then the set of observable behaviors of P' is a subset of that of P if:*

$$Reord(e, d) \ \wedge \ (\forall C_{i \in [1,n]}, \forall k \in C_i \ s.t. \ e \overrightarrow{ao} k \wedge k \overrightarrow{ao} d, \ Reord(e, k) \wedge Reord(k, d))$$

The above condition can be simplified as we already have corollary to show reordering of non-consecutive events No Candidate of P exists such that only one of e or d exists in them.

$$\nexists C \in P \ s.t. \ (e \in C \ \wedge \ d \notin C) \ \vee \ (e \notin C \ \wedge \ d \in C)$$

Proof. The proof would go as follows:

- By property of conditionals, there could be a candidate with e existing but d not. This would mean that d is a part of conditional brnaching. Meaning if d is event in one branch and k is in the other branch, then Property of conditionals must hold for any candidate.
- After reordering the property for candidates based on original program does not hold, thus having a candidate where d and k exist.
- If d is a write, then a new reads from relation can exist, thus introducing a new observable behavior.
- If d is a read, then too a new reads-from relation can exist.
- Note that here, we are talking in terms of candidates, meaning if a reads-from relation cannot exist, it is not due to the Consistency rules, but that such an event does not exist. This is different from restricting behaviors, rather its introducing new behaviors.
- Perhaps this should be mentioned in the proofs where only one candidate is considered too.

□

It has come to my notice that in general reordering may not be fine, if we end up removing something outside a conditional. To show this I have a simple counter example. However, the only way to show this, is to say that there is some candidate which suddenly has an agent order relation between two events which were not supposed to have any relation to them. Simply put, we can say that there is a candidate execution of the reordered program, where both the events exist, where as there isn't any candidate of the original program where such a thing can happen.

4.6

5 Elimination

There are two types of elimination we are concerned with:

- Read Elimination
- Write Elimination

Theorem 2.2. *Consider a candidate C of a program and its possible Candidate Executions where $\overrightarrow{hb}$ is strictly partial order. Consider an event e which is a read. Consider another Candidate C' without the event e. If e has an unordered access mode, then the set of Observable behaviors of C' is a subset of C without the relation $e \overrightarrow{rf} w$ where w is some write event in C.*

Proof. We look at this as an elimination of e that takes place in any candidate execution of C. We then go about answering the same four questions as we did for reordering. The only major change here being that elimination removes $\overrightarrow{hb}$ relations. We must check whether the removal of these relations introduce new behaviors, in contrast to that in reordering, where new relations were introduced.

1. Preserving *happens-before* relations The relations we want to preserve are those that are dervied through relation with e, meaning the following two relations:

a) $k \overrightarrow{hb} e$ b) $e \overrightarrow{hb} k$

We can divide the events involved in the above into two sets:

$$K_b = \{k \mid k \overrightarrow{hb} e\}.$$
$$K_a = \{k \mid e \overrightarrow{hb} k\}.$$

Put a figure here for an intuitive understanding of the problem at hand

We need to ensure the following relations hold after elimination.

$$\forall k_a \in K_a \ \wedge \ \forall k_b \in K_b \ . \ k_b \overrightarrow{hb} k_a \tag{5}$$

Slight notational confusion WHat if the eliminated event is a conditional check? That would mean events in the conditional check are also eliminated. Which would mean one has to check if it is okay to eliminate all events within the conditional.

Similar to reordering, we need to have a valid pivot pair $< p_b, p_a >$ such that

$$\forall k_b \neq p_b \in K_b \ . \ k_b \overrightarrow{hb} p_b \tag{6}$$
$$\forall k_a \neq p_a \in K_a \ . \ p_a \overrightarrow{hb} k_a \tag{7}$$

By Lemma 1, $e : uo$ is the only condition that satisfies our requirement. By Lemma 2, $e : uo \ \vee \ e : sc$ are the options. Condsidering both the above conditions to be satisfied, $e : uo$ is the only possibility that holds.

Write an expression which is the conjunction of both lemmas, and show how the conjunction boils down to the result that we come to.

2. The *happens-before* relations lost The relations lost are those attached to the event e, which are:

$$k \overrightarrow{hb} e \ \vee \ e \overrightarrow{hb} k \tag{8}$$

Do we need to prove that these are the only relations lost? Proof part 1 implicitly shows this.

3. Presence of Cycles? Because no new $\overrightarrow{hb}$ relations are introduced, and because original candidate executions have $\overrightarrow{hb}$ as a strict partial order, no cycles are introduced after elimination.

Perhaps write this argument a bit better.

4. Do the lost relations result in New Observable Behaviors? To answer this, we need to see whether the relations removed had an impact on $\xrightarrow[rf]{}$ relations other than those with e. To prove that it does not have any impact, we divide our argument into two parts, viz. into the two types of relations removed:

a) $k \xrightarrow[hb]{} R_u o$ b) $R_u o \xrightarrow[hb]{} k$

In the first case, we have the following possibilities.

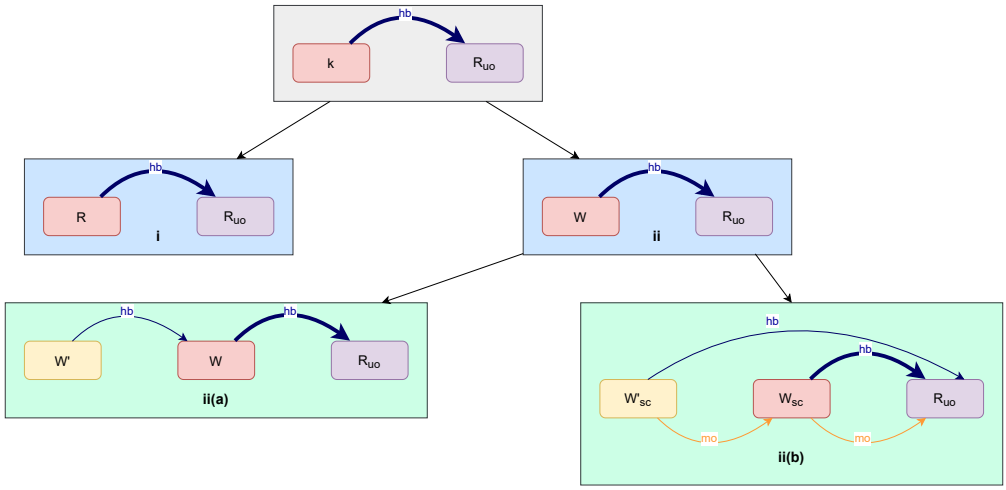

Figure 36: The first type of relations removed and the various patterns forbidden by them.

Observations:

- (i) is not a pattern forbidden by the consistency rules
- (ii)(a) is a pattern in Coherent Reads, however, only restricting $\xrightarrow[rf]{}$ relation with R and W'(which here is our Unordered Read)
- (ii)(b) is a pattern in Sequentially Consistent Atomics, however, once again only restricting $\xrightarrow[rf]{}$ relation with R and W'.

In the second case, we have the following possibilites.

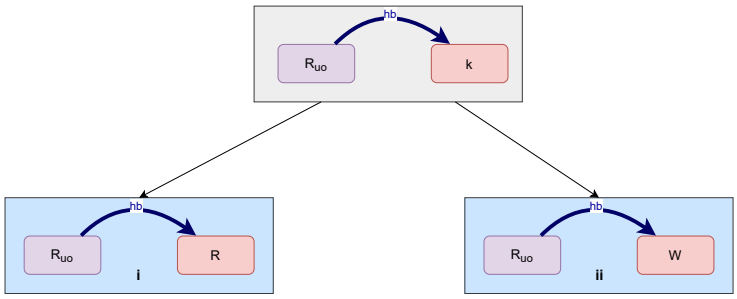

Figure 37: The second type of relations removed and the various patterns forbidden by them.

Observations:

- (i) is not a pattern in any Consistency rules
- (ii) is a pattern in Coherent Reads, however, only restricting $\xrightarrow[rf]{}$ relation with R and W

From the above observations, we can see that the relations removed only have restriction on reads-from relations on the event we elmiinate. Thus, by case wise analysis we can conclude that no new observable behaviors are introduced due to the removed $\xrightarrow[hb]{}$ relations.

□

Explain here why we consider two consecutive write events only. The argument being the Coherent Reads pattern can be triggered anyhow.

Theorem 2.3. *Consider a candidate C of a program and its possible Candidate Executions where $\xrightarrow[hb]{}$ is strictly partial order. Consider two **write** events e and d such that $cons(e, d)$ is true in C and $e \xrightarrow[ao]{} d$. Consider a Candidate C' without event e. If e has an unordered access mode and e and d have the same range, then the set of Observable behaviors of C' is a subset of C.*

Proof. Once again, we look at this as a write elimination done on a Candidate Execution of C. We start by proving when other happens-before relations remain intact. Followed by identifying relations lost due to elimination and a proof for when these relations do not introduce new observable behaviors.

Preserving Happens-before relations The relations we want to preserve are those that are dervied through relation with e, meaning the following two relations:

a) $k \xrightarrow[hb]{} e$ b) $e \xrightarrow[hb]{} k$

We can divide the events involved in the above into two sets:

$$K_b = \{k \mid k \xrightarrow[hb]{} e\}.$$
$$K_a = \{k \mid e \xrightarrow[hb]{} k\}.$$

Put a figure here for an intuitive understanding of the problem at hand
We need to ensure the following relations hold after elimination.

$$\forall k_a \in K_a \ \wedge \ \forall k_b \in K_b \ . \ k_b \xrightarrow[hb]{} k_a \tag{9}$$

Slight notational confusion
Similar to reordering, we need to have a valid pivot pair $< p_b, p_a >$ such that

$$\forall k_b \neq p_b \in K_b \ . \ k_b \xrightarrow[hb]{} p_b \tag{10}$$
$$\forall k_a \neq p_a \in K_a \ . \ p_a \xrightarrow[hb]{} k_a \tag{11}$$

By Lemma 1, $e : uo$ is the only condition that satisfies our requirement. It can be our p_a and by Lemma 2, $e : uo \ \vee \ e : sc$ are the possibilities. Condsidering both the above conditions to be satisfied, $e : uo$ is the only possibility that holds.
Again, show the conjuction of both conditions

2. The *happens-before* relations lost The relations lost are those attached to the event e, which are:

$$k \xrightarrow[hb]{} e \ \vee \ e \xrightarrow[hb]{} k \tag{12}$$

Do we need to prove that these are the only relations lost? Proof part 1 implicitly shows this.

3. Presence of Cycles? Because no new $\xrightarrow[hb]{}$ relations are introduced, and because original candidate executions have $\xrightarrow[hb]{}$ as a strict partial order, no cycles are introduced after elimination.
Perhaps write this argument a bit better.

4. Do the lost relations result in New Observable Behaviors? To address this, we divide our cases into two parts; one for each type of relation lost:

a) $k \xrightarrow[hb]{} e$ b) $e \xrightarrow[hb]{} k$

For the first case, we have the following possibilities:

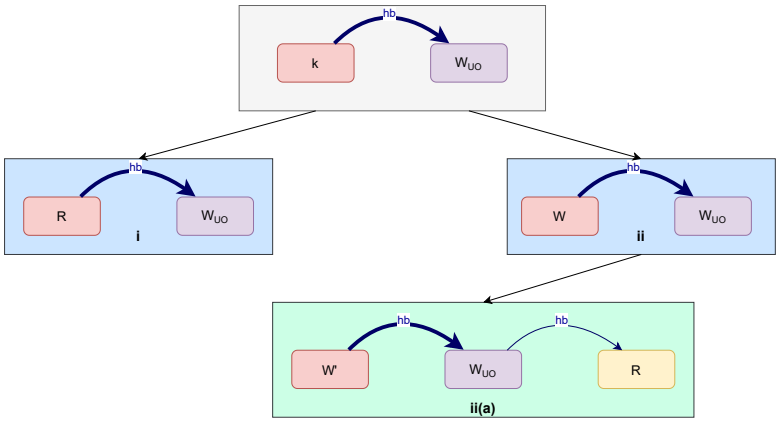

Figure 38: First case possibilities (change caption stimiar to that for read elim)

We can observe the following:

- (i) is a pattern from Coherent Reads that restricts the read R reading from W. And this will remain the case even after elimination of W.
- (ii)(a) is a pattern from Coherent reads, forbidding R to read from some W'. This will remain the case after elimination of W if firstly we have $d \xrightarrow{hb} R$. By Lemma 2 this is indeed the case. Secondly, we need to ensure that after elimination, the Coherent Reads pattern with d now restricts the exact set of $\xrightarrow{rbf}$ relations. Since we have no certain information on the range of R or W', we require the ranges of e and d to be same for our requirement to hold in general.
- **PErhaps explain the above argument in more detail**

For the first case, we have the following possibilities:

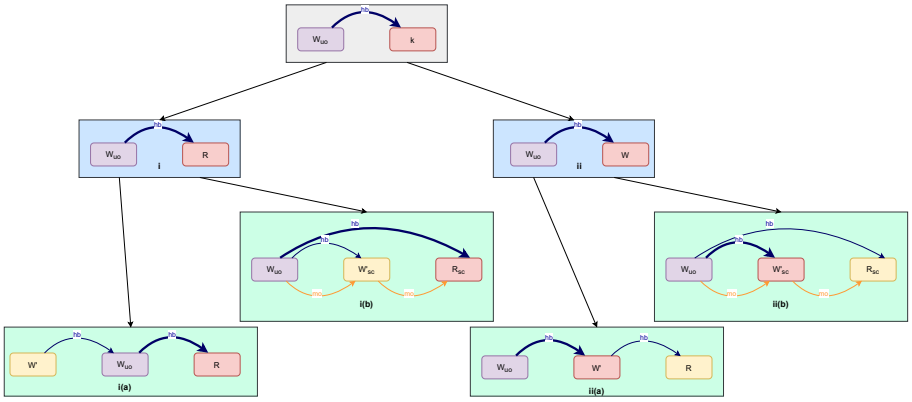

Figure 39: Second case possibilities (change caption stimiar to that for read elim)

We make the following observations:

- (i)(a) has the similar argument to the previous case's (ii)(a), requiring e and d to have equal ranges.
- (i)(b) is a pattern of Sequentially Consistent Atomics, which restricts R from reading anything of W. This will reamin the case after W is eliminated.
- (ii)(a) is a pattern of Coherent Reads, restricting R from reading W. This will remain the case after eliminating W.
- (ii)(b) is the same as (i)(b), hence the argument remains the same.

In all the above cases, observe that on keeping range of e and d equal, none of the patterns introduce any new observable behavior. Hence, if we have two consecutive writes of equal ranges, of which the

first one has access mode unorderd, the set of Observable Behaviors without the write is a subset of that with it present.

□

Corollary 2.3.1. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider two write events e and d both having equal ranges such that* $\neg cons(e, d)$ *is true in C and* $e \xrightarrow[ao]{} d$*. Consider another Candidate C' without the event e. Then, the set of Observable behaviors possible in C' is a subset of C only if the following holds true.*

$$\forall\ i\ \in\ [1, n-1]\ s.t.\ cons(e, k_1)\ \wedge\ cons(k_n, d,\ )\wedge\ cons(k_i, k_{i+1}, ,)$$
$$\exists\ (n+1) \geq j > 0\ s.t.\ \forall l\ \in\ [1, j-1]\ .\ Reord(e, k_l)\ \wedge\ \forall m\ \in\ [j, n]\ .\ Reord(k_m, d)$$

Proof. We prove this using induction on the number of events k between e and d. For each case, we see whether a valid j exists.

Base Case (n=1) For this case, if we have $Reord(e, k1)$ then $j = 2$ is a valid choice. By Theorem of reordering, we get Candidate C'' with $cons((, e), d)$ whose observable behaviors are a subset. By Theroem (write elim), a observable behaviors of C' is a subset of that of C''. By transitivity property of subsets, behaviors of C' is a subset of C.

Write above arguments properly

While if we have $Reord(k1, d)$ then $j = 1$ is a valid choice. The argument is the same as above.

Inductive Case (n) Suppose the corollary holds for the case n. Meaning, the observable behaviors of C' is a subset of C. And suppose j is alos the number as needed.

Then for the case where there are $n + 1$ events,we have the following two cases:

If k_x is the additonal event added in between e and d, then, if $Reord(k_x, d) \wedge x > j$, the new j remains the same as the old one. Because $Reord(K_{n+1}, d)$, by theorem of reordering, the Candidate C'' after reordering has observable behaviors as a subset of C. Now after reordering, we have our inductive case assumption, hence observable behaviors of C' is a subset of C.

On the other hand, if $Reord(e, k_x) \wedge x \leq j$, the new j is plus one the old j. Because $Reord(e, k_1)$ by theorem of reordering e and k_1, the Candidate C'' after reordering has observable behaviors as a subset of C. Now j for C'' becomes $j - 1$, hence we get our original inductive case assumption on n. By transitive property of subsets observable behaviors of C' is a subset of C.

Very rudimentary format of arguments, discuss with Clark and get them more formal

□