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

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 ($\overrightarrow{ao}$) 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 ($\overrightarrow{sw}$) 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.

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

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

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

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 $\overrightarrow{ao}$ 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 1.*

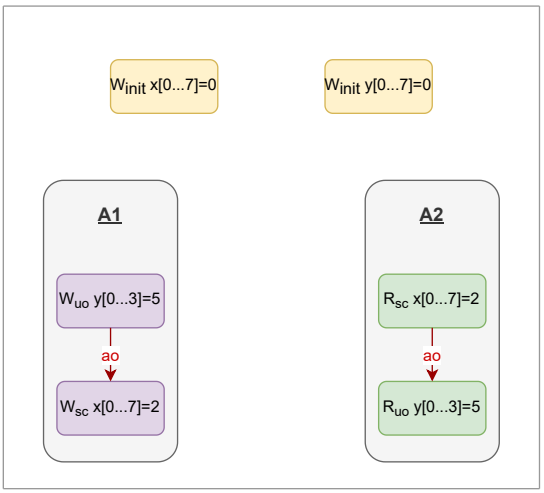

Figure 1: An example of a Candidate

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.*

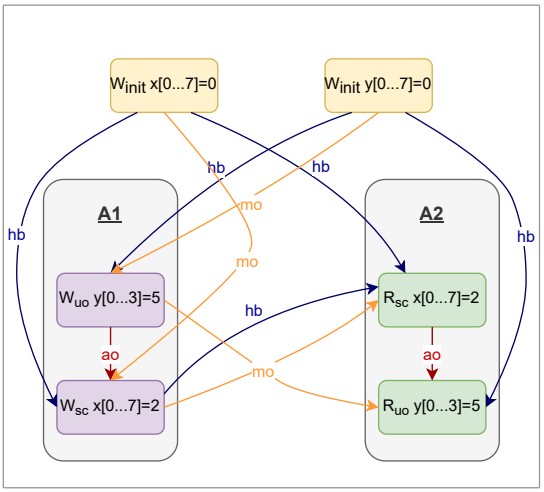

Figure 2: An example of an Execution based on Candidate above

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.*

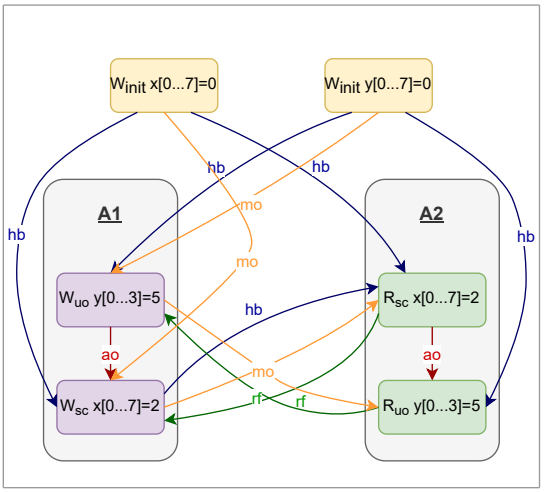

Figure 3: Observable Behavior

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.

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 $\overrightarrow{hb}$ 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.

5 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).*

6 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.

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 $\overrightarrow{hb}$ 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.

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* $\overrightarrow{ao}$ *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* $dir(e,d) \ \Rightarrow \ cons(e,d) \ \vee \ e \overrightarrow{sw} d$

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

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

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

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

An event that satisfies the above with d is e.

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

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

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

The following figure should explain this intuition:

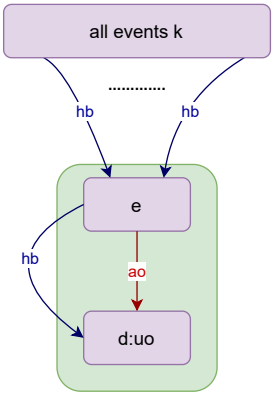

Figure 4: For the first case

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 $k \neq e$

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

The following figure explains this intuition:

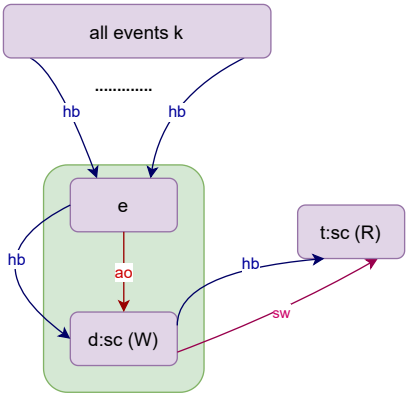

Figure 5: For the second case

□

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

If

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

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

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

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:

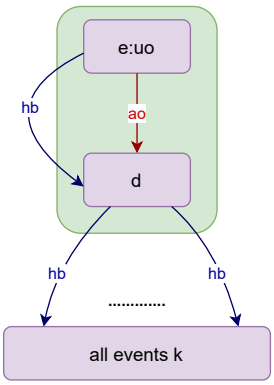

Figure 6: Caption

In the second case,

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

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:

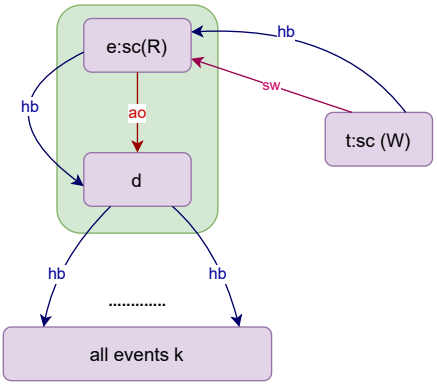

Figure 7: Caption

□

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.*

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

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

7.4 From Candidates to Program

Insights:

- At the program level, what is done in general is code motion.

- We can classify differnt cases of code motion.
- The two parts would be that of conditionals and that of loops.
- We still did not define general reordering, but we viewed it in terms of agent order to be general enough.

General Corollary defining code motion in a Candidate, not a program.

Corollary 2.0.1. *Consider a Candidate C of a program and its Candidate Executions which are valid. Consider a set of events $k_1, k_2, ...k_n$ such that*

$$\forall i \in [1, n-1] \ . \ k_i \overrightarrow{_{ao}} k_{i+1} \wedge cons(k_i, k_{i+1}) \tag{1}$$

Consider an event e such that

$$cons(e, k_1) \ \wedge \ e \overrightarrow{_{ao}} d \tag{2}$$

Consider another candidate C' with the only differnence from C being that

$$cons(e, k_n) \ \wedge \ k_n \overrightarrow{_{ao}} e \tag{3}$$

Then the set of observable behaviors of C' is a subset of that of C only if

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

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

The above corollary we defined is the most 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.

Make the corollary more precise, it can be simplified greatly.

Preliminaries needed for conditionals:

- One conditional results in two possible candidates. Describe formally that the two candidates are distinct, in that there are some events that aren't present in each of them, but are present in the other.
- There are two types of conditionals; need to have two defintions for conditionals then.

Perhaps we need a general corollary for program level

Corollary 2.0.2. *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 the following conditions hold:*

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

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.
- Reordering would then result in removing certain agent order relations with e and adding the same with d.
- Removal occurs because after reordering, P' will have candidates with d existing but not e.

- Adding occurs in the same fashion.
- Removal in this case may result in new observable behaviors with events outside the same agent
- Adding may also introduce new observable behaviors with events outside the same agent.
- The new observable behaviors can occur due to the other conditional branch having some sc events that syncronize with other threads.
- Maybe finding SC events in the other branch may not be so difficult to do. But perhaps if it is difficult, then reordering should not be done.
- Perhaps we could say that those events with whome d has now relations, if it is the case that it has a new relation with some k thats of type sc (perhaps we can narrow it down to a read / write), then we shouldnt be allowed to do reordering.

□

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.

Proof.
- We can easily show with our corollary of reordering non consecutive events that the above conditions must hold.
- The second part of the proof would involve going into two parts, one for conditional and one for loops
- The part of conditionals we can show by counter example, that a new observable behavior can occur, thus making our subset claim false.
- The part for loops may not be right actually, we have to change our statement once again as we would need to show that there isnt a set of events k and e or d in any original candidate where loops weren't iterated. But e or d exists in some candidate execution of the reordered program.

□

7.5

8 Elimination

There are two types of elimination we are concerned with:

- Read Elimination
- Write Elimination

Theorem 2.1. *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) $k \xrightarrow[hb]{} e$ 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.

$$\forall k_a \in K_a \ \wedge \ \forall k_b \in K_b \ . \ k_b \xrightarrow[hb]{} k_a \tag{5}$$

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

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

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:

$$k \xrightarrow[hb]{} e \ \vee \ e \xrightarrow[hb]{} k \tag{8}$$

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 $\xrightarrow[hb]{}$ 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 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.

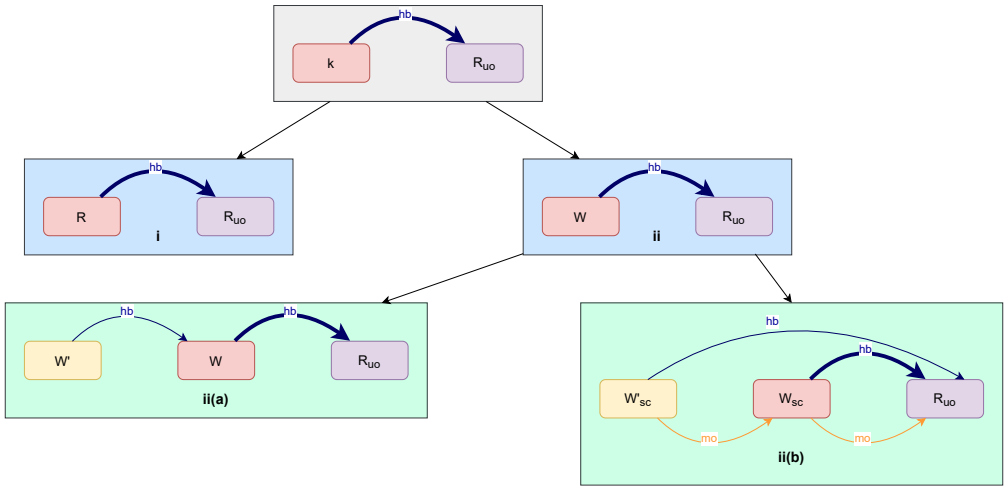

Figure 8: The first type of relations removed and the various patterns forbidden by them.

Observations:

- (i) is not a pattern forbidden by the consistency rules
- (ii)(a) is a pattern in Coherent Reads, however, only restricting $\overrightarrow{rf}$ 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 $\overrightarrow{rf}$ relation with R and W'.

In the second case, we have the following possibilites.

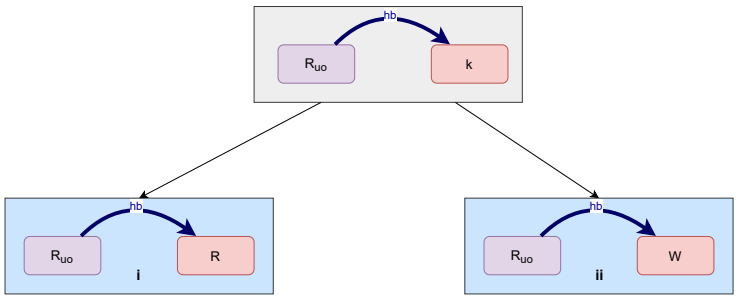

Figure 9: The second type of relations removed and the various patterns forbidden by them.

Observations:

- (i) is not a pattern in any Consistency rules
- (ii) is a pattern in Coherent Reads, however, only restricting $\overrightarrow{rf}$ 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 $\overrightarrow{hb}$ relations.

□

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

Theorem 2.2. *Consider a candidate C of a program and its possible Candidate Executions where $\overrightarrow{hb}$ is strictly partial order. Consider two **write** events e and d such that $cons(e, d)$ is true in C and $e \overrightarrow{ao} d$. 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:

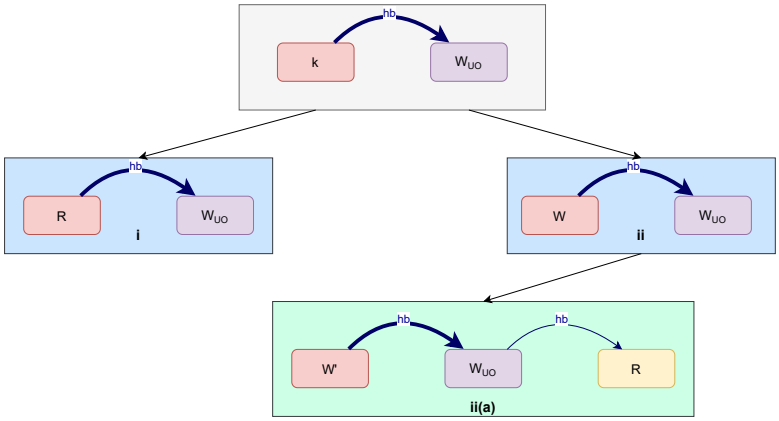

Figure 10: First case possibilities (change caption stimiar to that for read elim)

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:

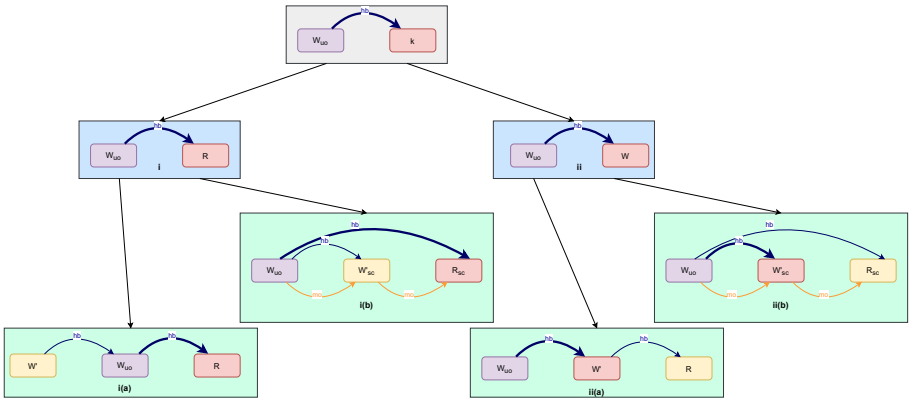

Figure 11: Second case possibilities (change caption stimiar to that for read elim)

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 2.2.1. *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

□