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.

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

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.

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.

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, 32}. This is because, we are taking decimal equivalent of a 32 bit binary number. It is important to note this fact.

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 for each shared memory event that tells us the kind of ordering that it respects.

1. **Sequentially Consistent** (*sc*) - Events of this type are *atomic* in nature. The meaning of sequentially consistent implies that there is a strict global total ordering of such events which is agreed upon by all concurrent processes sharing the same memory.
2. **Unordered** (*uo*) - Events of this type are considered non-atomic and can occur in different orders for each concurrent process, meaning 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. Additionally, only write events can be of this type and there is only one init event for each byte address in shared memory.

We represent the type of events in the following format - *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 specific hardware, such guarantees are satisfied.

The notion of sequentially consistent has the same semantics of what C++ has for such events. Note that this semantics is not mentioned here explicitly, but by talking with fellow researchers working on the same domain, as well as with careful observation, it has come to our understanding that this is assumed to be true. We will make sure that the semantics is defined here properly as per what is there of C++.

We are not sure 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 not mentioned in the *standard*. We assume them to be for individual byte addresses, as we will see shortly in the rule for $\overrightarrow{hb}$ ordering why we consider this assumption to be the best one. But note that it may not be the case always. As an example, consider a 32-bit hardware not having instructions to support 16 or 8 bit memory actions. In that case we do not know what the *init* event will sum up to. It would rather be better for it to be just one *init* write event that ranges through the entire shared memory, and modify the $\overrightarrow{hb}$ relation accordingly.

1.5 Tearing (Or not)

Additionally, each shared-memory event is also associated with whether they are tear-free operations or not.

Tearing Operations that tear are not aligned accesses with respect to the hardware and can be serviced using two or more memory fetches.

Tear-Free Operations that are tear-free are aligned with respect to the hardware and should appear to be serviced in one memory fetch. (this might not be possible always, but we are concerned with whether it can appear to be tear free)

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. This is also defined as a set of axioms further below. We make this distinction to avoid confusion :

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.

For most of our analysis, unless otherwise stated we will assume all events to be tear-free

2 Relation among events

There are three basic relations that assist us in reasoning about events and their interaction with memory.

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 would be:

$$r \overrightarrow{rbf} \{(w_1, i), (w_2, i+1), (w_2, i+2)\}$$

We will represent individual rbf relations with read events in our examples. So for the above example, the three rbf pairs are:

$$r \overrightarrow{rbf} (w_1, i),\ r \overrightarrow{rbf} (w_2, i+1),\ r \overrightarrow{rbf} (w_2, i+2)$$

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 :

$$r \overrightarrow{rf} \{w_1, w_2\}$$

Similar to rbf, we also represent pair-wise relation in rf :

$$r \overrightarrow{rf} w_1,\ r \overrightarrow{rf} w_2$$

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:

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

A property that must hold for each of these lists is:

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

$$\forall i, j > 0,\ \langle s^i, s^j\rangle \in ASW_j \Rightarrow i \neq j \quad \wedge \quad s^j \in ael(k)$$

The analogy is similar to the property that every unlock must be paired with a subsequent lock, which enforces the condition that a lock can be acquired only when it has been released.

3 Ordering Relations among Events

Agent Order ($\xrightarrow{ao}$) An relation on all events. It respects the order in which the events are evaluated by a particular agent.

$$\forall i, j > 0,\ e^i_j \xrightarrow{ao} e^i_{j+1}$$

1. Agent order also respects transitivity.

$$(e \xrightarrow{ao} d) \wedge (d \xrightarrow{ao} g) \Rightarrow (e \xrightarrow{ao} g)$$

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 and meant to be equivalent to what we call as intra-thread sequential order. It is the same as what **sequenced-before** is defined to be in C++

Synchronize-With Order ($\xrightarrow{sw}$) An ordering relation that represents the synchronizations among different agents.

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

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

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

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

Note that for the second condition, both ranges of events have to be equal. However, there is no restriction that the read event cannot read-from other write events.

Happens Before Order ($\xrightarrow{hb}$) A *partial order* on events which are composed of the following:

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

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

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

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

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

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

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

Memory Order ($\xrightarrow{mo}$) This order is a *total order* on all events that are evaluated in an agent cluster.

1. Happens before order is a part of memory order

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

4 Valid Execution Rules (the Axioms)

We use the above concepts in reasoning about executions of a given concurrent program. Each execution is called as a candidate. Each candidate execution must satisfy the rules that we will specify below to be declared a *valid* execution.

Coherent Reads There are certain restrictions of what a read event can see at different points of execution based on $\overrightarrow{hb}$ relation with write events.

- A read(R) cannot read a value from a write(R) that has happened after it !

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

- A read(R) cannot read a specific byte address value from write(W) if:
  - W is bound to have happened before R
  - There is a write(V) that happens between them which modifies the exact byte that R reads from W.

  Note that this rule would be on the rbf relation among two events.

$$W \overrightarrow{hb} R \ \wedge \ W \overrightarrow{hb} V \ \wedge \ V \overrightarrow{hb} R \ \Rightarrow \ \forall x \in (\Re(W) \cap \Re(V) \cap \Re(R)), \ \neg \ (R \overrightarrow{rbf} (W, x))$$

The reason why we focus on the negation is because defining what could be seen by a read event is a non-trivial task to define compared to what cannot be seen. The negation helps better understand what not to assume about your program when you want to reason about it's possible execution outcomes.

Tear-Free Reads If two tear free writes V and W and a tear free read R all with equal ranges exist, then R can read only from one of them[1]

$$W\!:\!tf \ \wedge \ V\!:\!tf \ \wedge \ R\!:\!tf \ \wedge \ (\Re(W) = \Re(V) = \Re(R)) \Rightarrow (R \overrightarrow{rf} W \wedge \neg (R \overrightarrow{rf} V)) \ \vee \ (R \overrightarrow{rf} V \wedge \neg (R \overrightarrow{rf} W))$$

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[2], we assume the following memory order among writes W and V and a read R to be the premise for all the rules:

$$W \overrightarrow{mo} V \overrightarrow{mo} R$$

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

$$W\!:\!sc \wedge R\!:\!sc \wedge R\!:\!sc \wedge \Re(W) = \Re(V) = \Re(R) \ \Rightarrow \ \neg R \overrightarrow{rf} W$$

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

$$W\!:\!sc \ \wedge \ V\!:\!sc \ \wedge \ \Re(W) = \Re(V) \ \wedge \ W \overrightarrow{hb} R \ \wedge \ V \overrightarrow{hb} R \Rightarrow \neg R \overrightarrow{rf} W$$

- If V and R are sequentially consistent, having equal ranges, and W is bound to happen before them, then R cannot read from W

$$V\!:\!sc \ \wedge \ R\!:\!sc \ \wedge \ \Re(V) = \Re(R) \ \wedge \ W \overrightarrow{hb} V \ \wedge \ W \overrightarrow{hb} R \ \Rightarrow \ \neg R \overrightarrow{rf} W$$

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

We think we do not necessarily need all ranges to be equal, this can be weakened by actually stating that the intersection of ranges is equal followed by refining the rule for $\overrightarrow{rbf}$ relation

Write events that are sequentially consistent are observed to happen in the same memory order by every agent. This is without any specific $\overrightarrow{hb}$ relation among such events. Additionally, $\overrightarrow{hb}$ relation is consistent with the memory order

Liveliness of Sequentially Consistent Writes The standard also enforces that every sequentially consistent write is eventually read. In that sense, we have:

$$d{:}sc \ \wedge \ d{\in}W \ \Rightarrow \ \exists d.\ d \ \overrightarrow{rf} \ W$$

5 Race

Race Condition *RC* We assume race condition to be a set of pairs of events that are in a race.

Two events e and d are in a race condition when they are shared memory events

$$(e \in \boldsymbol{SM}) \wedge (d \in \boldsymbol{SM})$$

having at least overlapping ranges, which are either two writes or the two events are involved in a $\overrightarrow{rf}$ relation with each other.

$$(e, d \in (\boldsymbol{W} \cup \boldsymbol{RMW}) \wedge (\Re(d) \cap_{\Re} \Re(e) \neq \phi)) \vee ((d \ \overrightarrow{rf} \ e) \vee (e \ \overrightarrow{rf} \ d)) \Rightarrow (e, d) \in RC$$

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

It is interesting to note that the standard specifies only those read-write events that do have a $\overrightarrow{rf}$ relation among them to be in a race condition, while the relation signifies an order that has resolved itself because the $\overrightarrow{rf}$ relation is there. According to me, their intention was to say that if there *could* exist a $\overrightarrow{rf}$ relation among two events, then they would definitely be in a race. This clarification needs to be incorporated in the axiom that defines what is a race condition.

Data Race *DR* Two events are in a data race when they are already in a race condition and when the two events are not of type *sc* together or they have overlapping ranges:

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

The definition for data race also implies that sequentially consistent events with overlapping ranges are considered to be in a data race. This is counter-intuitive in the sense that if all agents observe the same order in which these events happen, then there is no question of it being in a data-race as such, there is a unanimous agreement among all agents about the order in which they take place during the execution.

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.

6 Consistent Execution

Valid Execution A valid execution is when the above valid execution rules are not violated, plus a few restrictions on the happens-before order[3]. So to summarize, a valid execution is when:

- $\overrightarrow{hb}$ is a strict partial order.
- Reads that exist are not
  - Invalid coherent reads.
  - Invalid tear-free reads.
- Execution does not violate sequentially consistent atomics.

The memory model guarantees that every program must have at least one valid execution

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, but it is interesting to note that this rule has implications of how to write compilers for specific hardware and that they are free to 'strengthen' the model if need be.

7 Instruction Reordering

One of the most common program transformations done by the compiler as well as the hardware is instruction reordering. The requirements for reordering in a sequential program is already well established and understood(cite). However, in a concurrent context, this is relatively less clear.

Simple reordering is not straightforward under shared memory semantics One of the main aspects of programs using shared memory is the aspect of visibility of a memory write to different threads. The relaxed memory model of Javascript prescribe semantics for this using the $\overrightarrow{hb}$ relations. Given this relation, it can be observed that seemingly "harmless" reordering using a sequential justification may not always keep the program behavior unchanged.

What can be done? A simple example-based analysis, though would work for small programs become infeasible as the programs scale in length and complexity. It is also important to note that generalizations by using a small sample size is not something we can afford especially when we want to ensure these program trasnformations are automated contrast to being transformed manually.

7.1 Our approach

One solution to this is to construct a proof that would expose/specify the conditions under which reordering is possible. To keep things simple, we first consider when consecutive instructions 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 exist in any execution of the original program. It is important to note however, that a proof in this sense would be generalized to any program , so often it might be the case that for specific programs, instruction reordering can be valid, however, in a general sense may not be valid for others.

7.2 Preliminaries

Before we go about proving when reordering is valid, we would like to have some definitions which would prove useful for going about the proof. These definitions are to create clear separation between a program and its possible executions. Having this separation will help us reason about reordering more clearly.

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 the code of the program involved in one possible execution abstracted to the set of shared memory events that are involved, with the added $\overrightarrow{ao}$ relations. 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.*

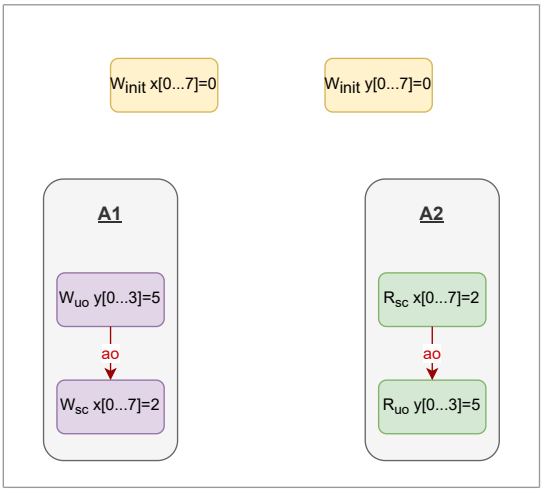

Figure 1: An example of a Candidate

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.*

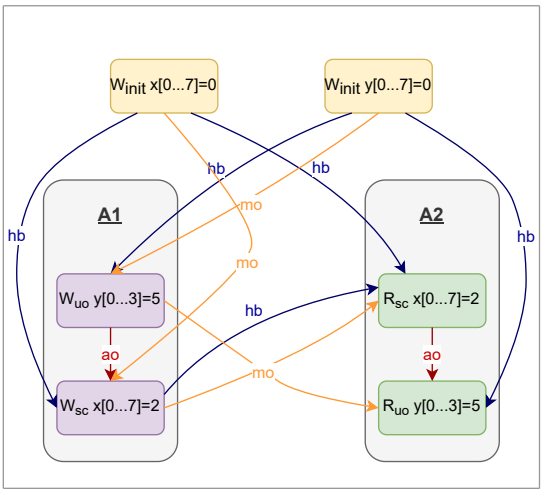

Figure 2: An example of an Execution based on Candidate above

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*

An Execution with $\overrightarrow{rf}$ *relations. This can be viewed as the outcome of a run of the Program, with the result as well as the witness to justify it.*

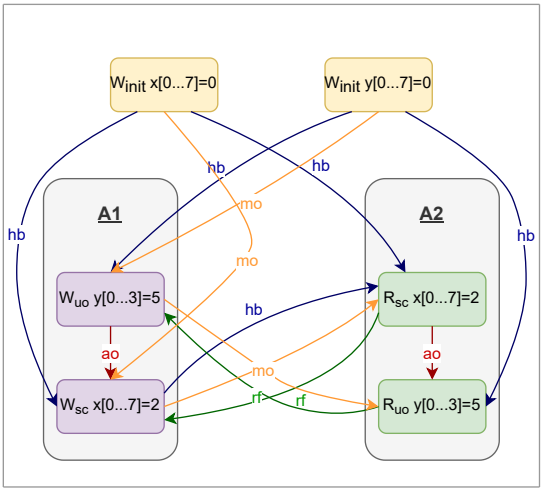

Figure 3: Observable Behavior

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.*

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*

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

Input space for cons *is the Cartesian product of Event set with itself*

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

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

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

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

7.3 Lemmas to assist our proof

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

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

If

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

then,

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

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

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

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

In the first case,

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

Then for any event k

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

The event that has such a relation with d is e.

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

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

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

The following figure should explain this intuition:

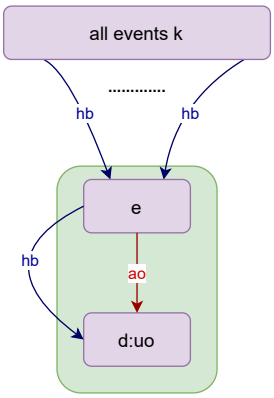

Figure 4: For the first case

In the second case,

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

Then for any event k

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

We once again have event e having such a relation

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

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:

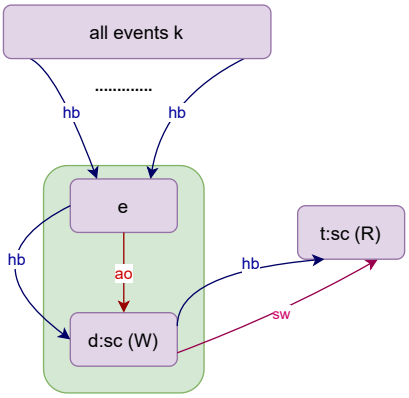

Figure 5: For the second case

□

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 has such a relation 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:

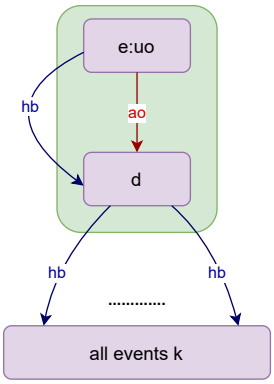

Figure 6: Caption

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 e having such a relation

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

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:

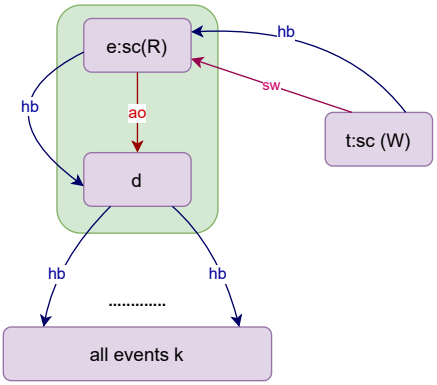

Figure 7: Caption

□

7.4 Abstract view of 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. The main relations we would want to retain would be those described by $\xrightarrow[hb]{}$. The intuition is that the axioms of the memory model rely mainly on this ordering relation to restrict observable behaviors in a program. It is important to note that this is a conservative assertion to ensure reordering is safe, and by no means does it say that one cannot reorder otherwise. There could be certain examples of programs that would not harm when certain reordering is done, but in a general sense, this may not be true for all possible programs.

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.

One might wonder what about the $\xrightarrow[mo]{}$ relation. The intuition for us is that memory order, though it plays a role in the observable behaviors, only influences the observable behaviors when considering all sequentially consistent events. (refer to the last axiom of the memory model). In addition to this, memory order plays a role only with events having equal range that conflict. Otherwise, every other axiom, necessarily relies on $\xrightarrow[hb]{}$ relation. Of course, the tear-free axiom does not involve any ordering relations, but since we are not changing the tearing aspect of events during reordering, that aspect does not concern our proof.

7.5 Valid reordering

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.*

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

Theorem 2.1. *Consider a candidate C of a program and its Candidate Executions which are valid. Consider two events e and d such that cons(e,d) is true in C and $e \overrightarrow{ao} d$. Consider another candidate C' resulting after reordering e and d. Then if* Reord(e,d) *is true in C, the set observable behaviors possible in C' is a subset of that of C.*

Proof. 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 then show that these new relations do not introduce any new observable behaviors. Lastly, we identify from the remaining cases whether *happens-before* cycles could be introduced.

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

1. Apart from the events involved in reordering, do other *happens-before* relations remain intact?
2. Are any new *happens-before* relations established?
3. Do they new relations bring new *observable behaviors?*
4. Are any *happens-before* cycles introduced?

1. Preserving *happens-before* relations If some $\overrightarrow{hb}$ relations among events are lost after reordering, we may introduce new observable behaviors. So the first step is to ensure that the existing relations in any Candidate Execution of C remain preserved after reordering, thus showing that the new Candidate Execution due to C' has the previous relations preserved.

The intuition is that if all the other $\overrightarrow{hb}$ relations are intact, the axioms that rely on this relation will not place more restrictions on possible $\overrightarrow{rf}$ relations.

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.

$$K_e = \{k \mid k \overrightarrow{hb} e\}$$

$$K_d = \{k \mid d \overrightarrow{hb} k\}$$

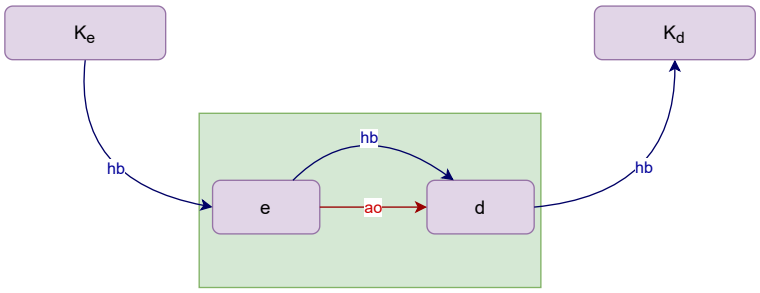

Figure 8: For any Candidate Execution of C, the set K_e and K_d

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

To ensure this, firstly note that in terms of direct happens-before relations, on reordering, any $CandidateExecution$ of C will have the following changes

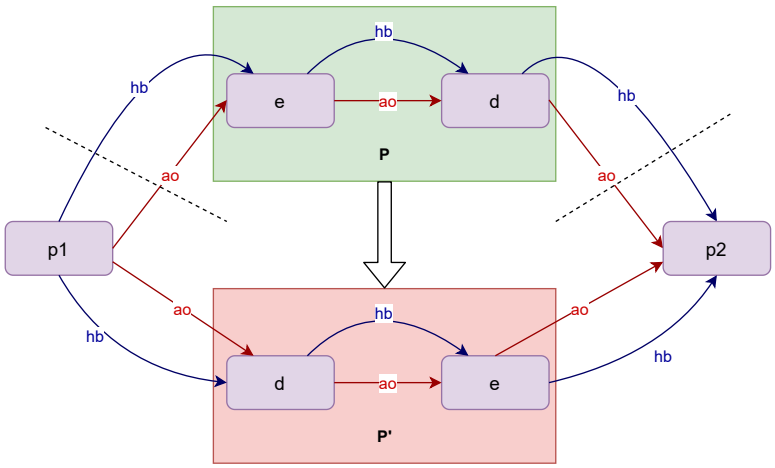

Figure 9: The direct relation changes that can be observed while reordering events e and d

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

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

and for that of C',

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

Note that here, $p1$ and $p2$ are part of the respective sets of events K_e and K_d. So we have the following relations that we need to preserve w.r.t these two events

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

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

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

When e is the first event or d is the last event, assume a dummy event that can act as $p1$ and $p2$ having the same relations.

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

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

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

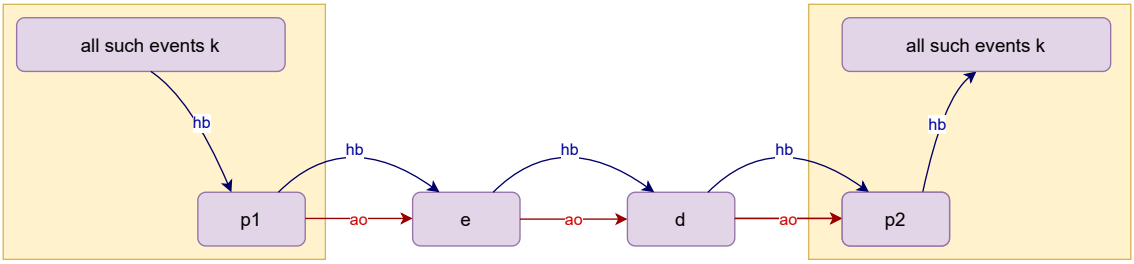

Figure 10: For any Candidate execution, the intuition behind valid pivots $< p1, p2 >$

By lemma 1, we have for C, the following condition where $p1$ is a valid pivot

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

Similarly, by lemma 2, we have for C, the following condition where $p2$ is a valid pivot

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

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

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

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

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

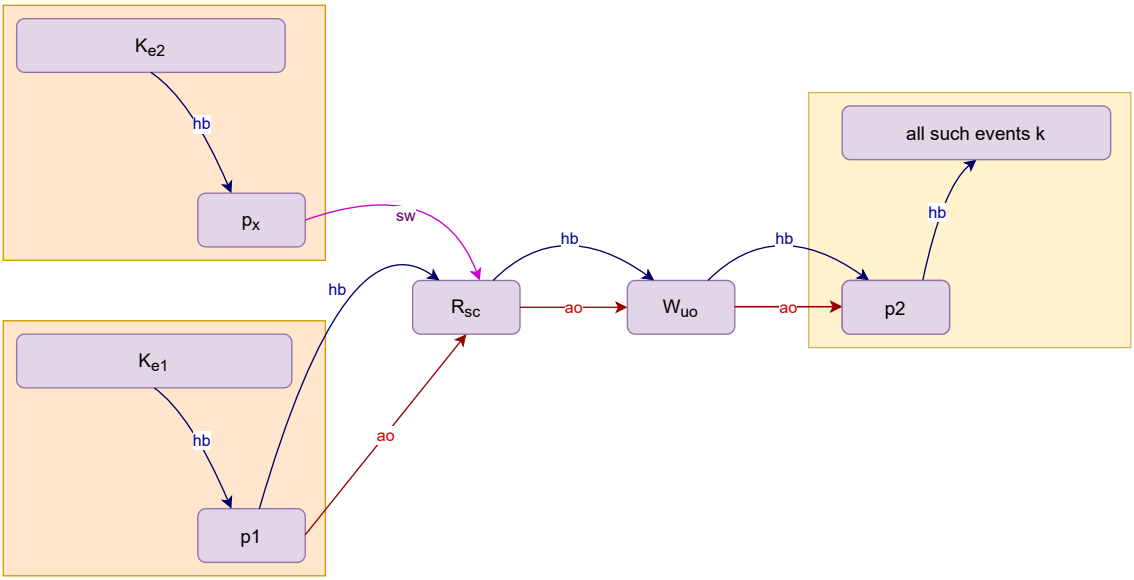

Figure 12: A Candidate Execution where p1 is not a valid pivot

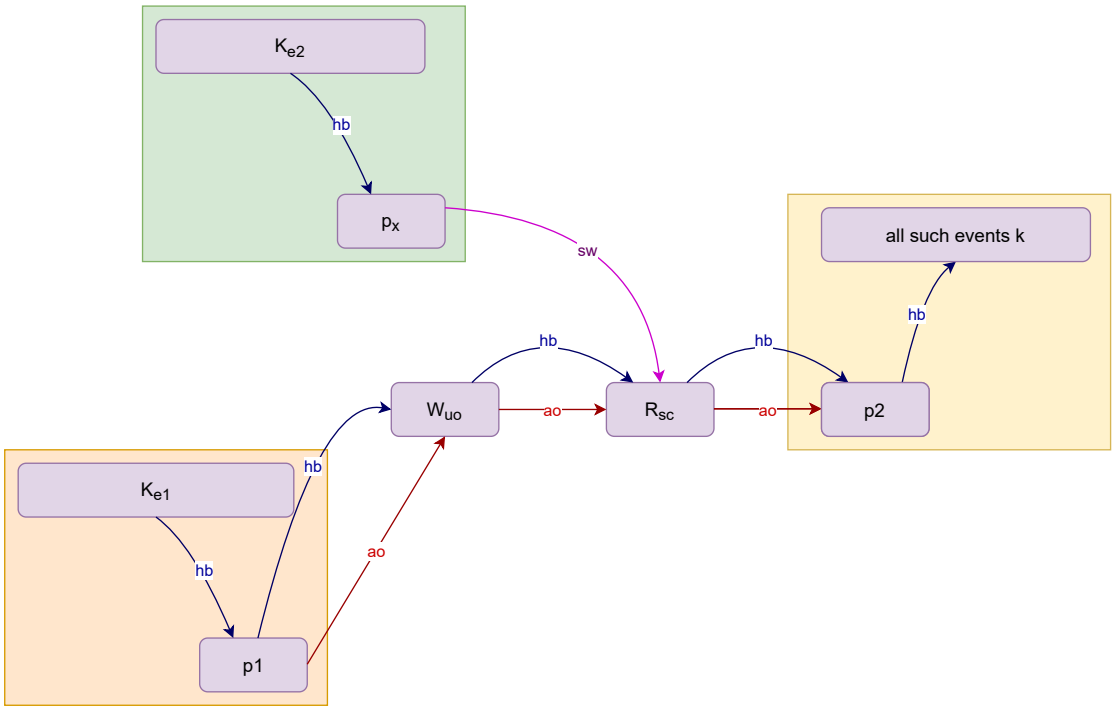

Figure 13: The resultant Candidate Execution after reordering, exposing the relations with p_x, K_{e2} and d that are lost

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

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

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

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

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

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

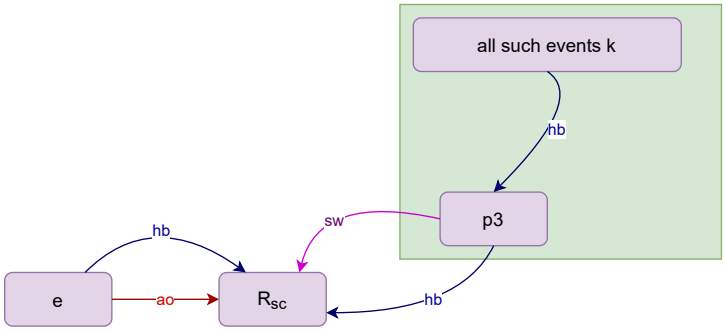

Figure 14: A Candidate Execution where d is a sequentially consistent read

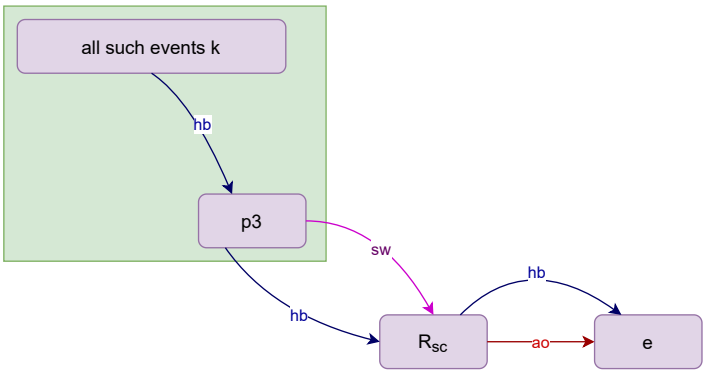

Figure 15: The Candidate Execution after reordering, exposing the new relations established with e, $p3$ and set k

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. Do new relations introduce new observable behaviors? In any candidate execution, reordering events e and d eliminate the relation $e \xrightarrow[hb]{} d$ and introduce the new relation $d \xrightarrow[hb]{} e$. If this new relation itself could bring about new behaviors, we need not consider other new relations introduced in the process.

To analyze this, notice that the axiom of *Coherent Reads* and that of *Sequentially Consistent Atomics* use $\xrightarrow[hb]{}$ relations to place restrictions on $\xrightarrow[rf]{}$. These are subject to the relation between the ranges of events involved. So, if we divide the cases we have based on e and d having disjoint, overlapping or equal ranges, we can summarize when these two axioms could play a role in allowing a new behaviour to occur.

Disjoint	R-R	R-W	W-R	W-W
uo-uo				
uo-sc				
sc-uo				
sc-sc				

Overlap	R-R	R-W	W-R	W-W
uo-uo				
uo-sc				
sc-uo				
sc-sc				

Equal	R-R	R-W	W-R	W-W
uo-uo				
uo-sc				
sc-uo				
sc-sc				

Figure 17: Cases based on the ranges of e and d where the axiom of Coherent Reads / Sequentially Consistent atomics can apply to introduce new observable behaviors on reordering

Notice that only when both e and d are reads, the relation of range in which they operate is irrelevant for any axiom to play a role. Whereas, in other cases, overlapping and equal ranges, the axioms do play a role.

For the other additional relations that are established, we can divide them into 5 cases.

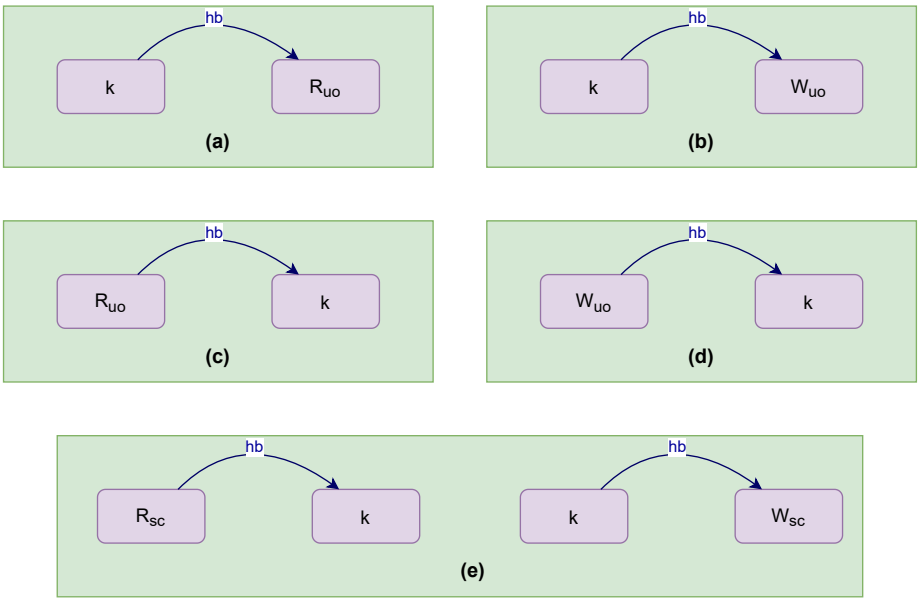

Figure 18: Caption

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

In the first case (a), depending on what event k is, we would have the following cases where the axioms would play a role.

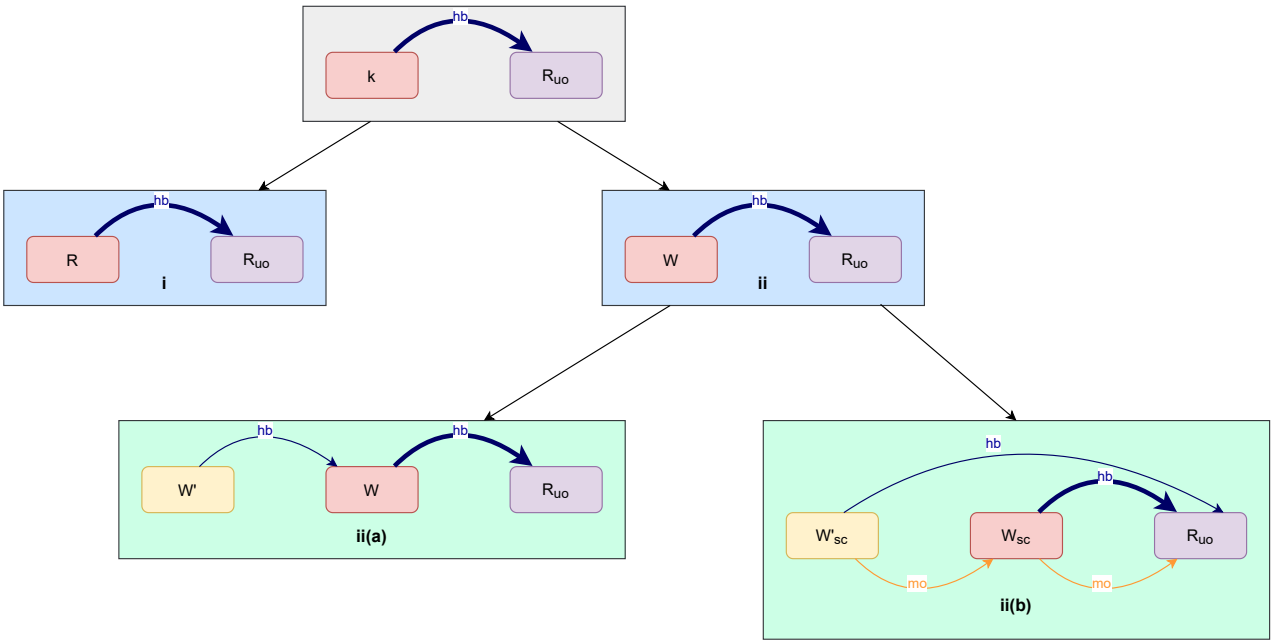

Figure 19: The role of the axioms on introducing a new relation between an unordered Read and some event k

1. For subcase (i), none of the axioms have any implications on observable behaviors.
2. For subcase (iii), the axiom of coherent reads could in restrict R from reading overlapping ranges of W' with W.
3. Subcase (iv) restricts R from reading W' by the axiom of sequentially consistent atomics if W and W' have equal ranges.

The other cases, also have instances which can satisfy some cases of the axioms, thus restricting possibly some $\overrightarrow{rf}$ relations.

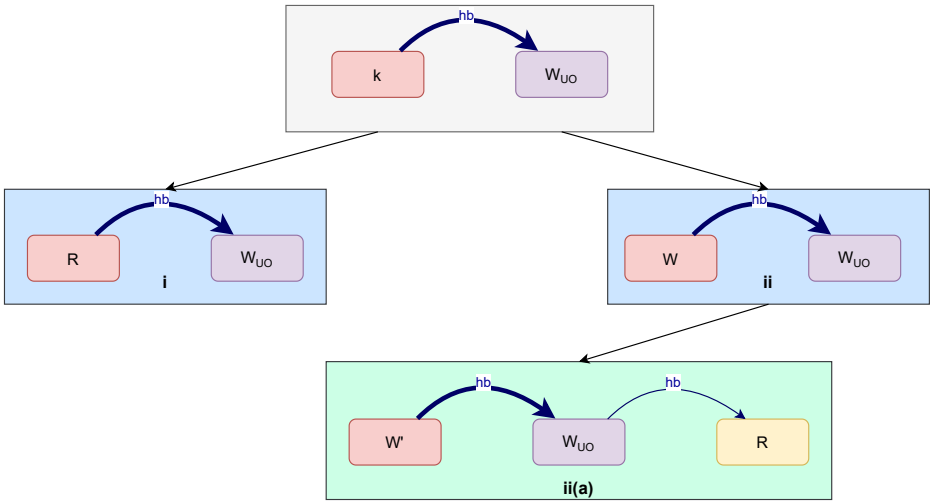

Figure 20: (i) and (ii(b)) satisfy the axiom of Coherent Reads

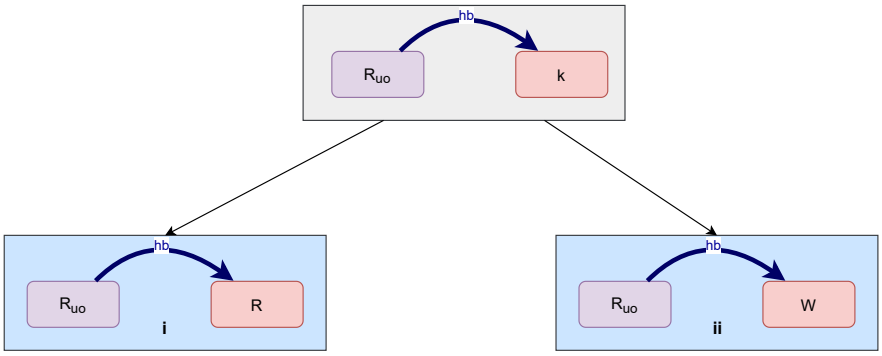

Figure 21: (ii) satisfies the axiom of Coherent Reads

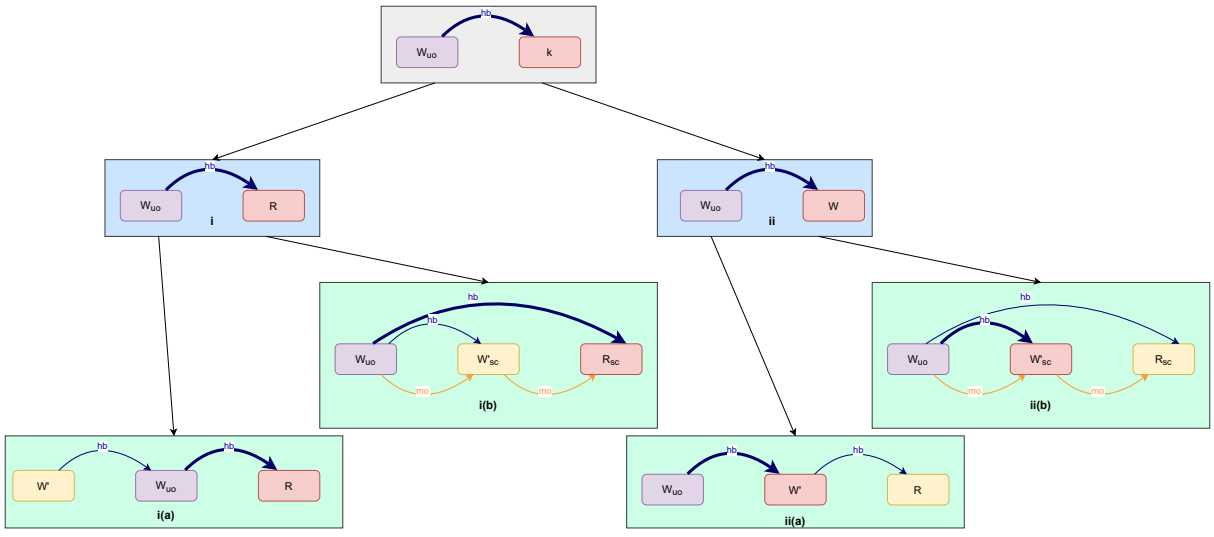

Figure 22: (i(a)), (ii(a)) satisfy the axiom of Coherent Reads, whereas (i(b)), (ii(b)) satisfy the axiom of Sequentially Consistent Atomics

The last case (v) is a repetition of cases of the above. The main point is to note that satisfying these particular axioms implies a restriction of $\overrightarrow{rf}$ relation among events. Note also that without this new relation, these conditions for these axioms wouldn't have been satisfied. Thus, this implies that we are only possibly reducing the observable behaviors of any Candidate Execution.

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

4. Presence of cycles? Though it may seem the previous questions have answered all our concerns to ensure set of observable behaviours shown by *Executions* of C' is a subset of C, we have not addressed whether the additional relations introduce any *happens-before* cycles. Consider the example of relations with three events below.

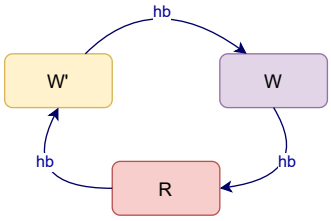

Figure 23: Caption

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. For our purpose, we assume the Candidate Executions of C contain no such cycles.

Note that if a cycle exists after reordering, then

1. The relations preserved do not themselves create a cycle
2. Additional new relations may introduce cycles

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

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

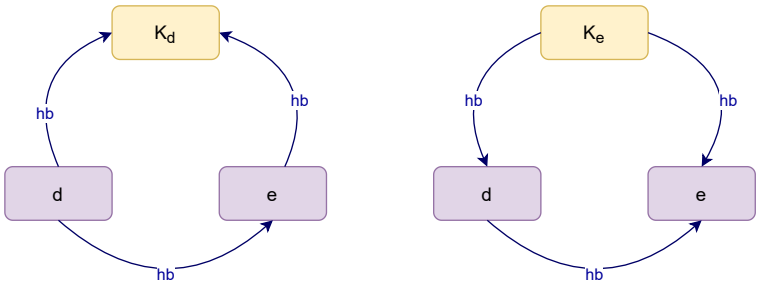

Figure 24: If k belongs to one of the sets K_e or K_d

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 $d \xrightarrow[hb]{} e$ does not exist. The following figure shows pictorially this fact.

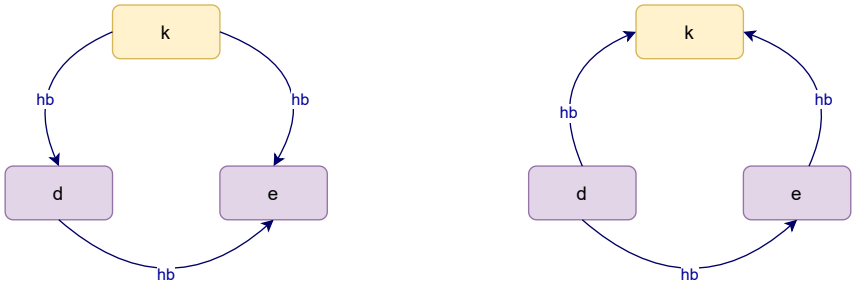

Figure 25: If $k \xrightarrow[hb]{} e$ or $d \xrightarrow[hb]{} k$ are new sets of relations

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.

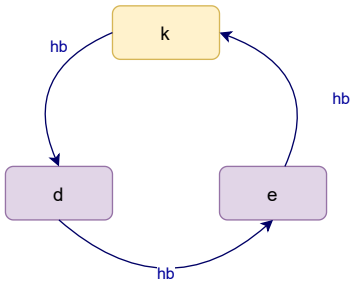

Figure 26: A cycle exists in the case where we have two new sets of relations ($k \xrightarrow[hb]{} e$ and $d \xrightarrow[hb]{} k$)

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

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

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

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

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

So, the following is also a cycle

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

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.

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

The table above, precisely is the definition of a reorderable pair.

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

□

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. □

7.6 Counter examples for the invalid cases

7.7