Equivalence Proof

Christopher Pulte, *22nd November 2018*

Theorem 1. *Let $C = (\mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{rmw})$ be a candidate execution. C is legal under ARMv8 Axiomatic if and only if there exists a trace of Flat Operational that induces* po*,* rf*,* co*, and* rmw*.*

The proof assumes a correct handling of the store exclusive dependencies in the axiomatic model (see remarks in the paper) — whereby a syntactic dependency out of the store exclusive's register write does not create memory model ordering from the load exclusive or the store exclusive. As before, the proof only covers finite executions, and only the behaviour of programs in which all memory accesses are not misaligned and of the same size (this also exclusive load-pair, store-pair; as the axiomatic model does not cover mixed-size accesses), and it does not cover the `LDAPR` weaker acquire instructions (which at the time of writing the proof was not covered by the operational model). The proof assumes that if a load and store exclusive instruction are successfully paired (rmw-related in the axiomatic model), then they are to the same address (see remarks in the paper). For simplicity, the proof assumes no reading from initial memory, i.e. that every memory read is satisfied from a write that originates from a store in the input program. Finally, the proof assumes the two models share the definition of the instruction semantics and thus agree on the definition of dependencies.

Hence, the assumptions are:

1. Finite candidate executions and finite Flat traces.
2. That if a write exclusive is successfully paired with a read exclusive, then they are to the same address.
3. ARMv8 Axiomatic and Flat Operational use the same instruction semantics.
4. $(E, E') \in \mathsf{addr}$ if and only if: E is a read and E' is a read or a write and E feeds into a register write that affects the address of E'. Note: this means $(E, E') \in \mathsf{addr}$ does not hold if E is a write exclusive.
5. $(E, E') \in \mathsf{data}$: E is a read and E' is a write and E feeds into a register write that affects the value written by E'. Note: this means $(E, E') \in \mathsf{data}$ does not hold if E is a write exclusive.

6. $(E, E') \in$ ctrl: E is a read and E' is a write and E feeds into a register write that affects a conditional branch or a computed branch instruction program-order before E'. Note: this means $(E, E') \in$ ctrl does not hold if E is a write exclusive, and that control dependencies are not delimited — ctrl; po $\subseteq$ ctrl.
7. All register reads of a load affect its address.
8. All register reads of a store affect its address and data.

The structure is as follows:

1. Section 1 (p. 2) gives the proof that ARMv8-axiomatic allows all behaviours allowed by the Flat operational model. First slightly simplifying the axiomatic model, then showing that for every candidate execution induced by Flat the main axiom external holds (in Section 1.1), then proving the intra-thread coherence axiom internal holds (Section 1.2), and last showing the atomic axiom holds (Section 1.3).
2. Section 2 (p. 13) defines the Flat-axiomatic intermediate model.
3. Section 3 (p. 18) proves that the Flat operational model allows the behaviours allowed by the Flat-axiomatic model.
4. Finally, Section 4 (p. 36) shows that the Flat-axiomatic model allows all behaviours allowed by the ARMv8-axiomatic model.

The statement then follows by Theorems 2 to 4.

1 Flat Operational behaviour included in ARMv8 Axiomatic

Let (po, rf, co, rmw) be a candidate execution. Ignoring weak acquire loads ("Q"), ARMv8 Axiomatic is the following:

```
let ca = fr | co
let obs = rfe | fre | coe
let dob = addr | data
        | ctrl; [W]
        | (ctrl | (addr; po)); [ISB]; po; [R]
        | addr; po; [W]
        | (addr | data); rfi
        | (ctrl | data); coi
let aob = rmw
        | [range(rmw)]; rfi; [A]
let bob = po; [DMB.SY]; po
        | [L]; po; [A]
        | [R]; po; [DMB.LD]; po
        | [A]; po
        | [W]; po; [DMB.ST]; po; [W]
        | po; [L]
        | po; [L]; coi
let rec ob = obs | dob | aob | bob | ob; ob
acyclic po—loc | ca | rf as internal
```

```
irreflexive ob as external
empty rmw \& (fre; coe) as atomic
```

This can be simplified for the purposes of the proof:

1. Since ca is used only in the definition of the internal axiom it can be inlined there.
2. Assume there is a cycle using an edge from [R];po;[dmb.ld];po. Then there is also one just using [R];po;[dmb.ld];po[R|W]: all aob|dob|bob edges are subset of program-order and therefore acyclic; and all edges in obs start from a read or write.
3. In the same way, replace [A];po with [A];po;[R|W],
4. po;[L] with [R|W];po;[L], and
5. po;[L];coi with [R|W];po;[L];coi.
6. The recursion in the definition of ob can be replaced by transitive closure.

With these simplifications, the model is equivalent to the one below:

```
let obs = rfe | fre | coe
let dob = addr | data
        | ctrl; [W]
        | (ctrl | (addr; po)); [ISB]; po; [R]
        | addr; po; [W]
        | (addr | data); rfi
        | (ctrl | data); coi
let aob = rmw
        | [range(rmw)]; rfi; [A]
let bob = po; [dmb.full]; po
        | [L]; po; [A]
        | [R]; po; [dmb.ld]; po; [R|W]
        | [A]; po; [R|W]
        | [W]; po; [dmb.st]; po; [W]
        | [R|W]; po; [L]
        | [R|W]; po; [L]; coi
let ob = (obs | aob | dob | bob)+
acyclic po—loc | fr | co | rf as internal
irreflexive ob as external
empty rmw & (fre; coe) as atomic
```

Lemma 1. *Let Tr be a valid trace of Flat Operational that induces the relations* po*,* co*,* rf*, and* rmw*. Then there is a valid trace Tr' that induces the same* po*,* co*,* rf*, and* rmw *relations but has no restarts and no discarded instruction tree branches ("always speculates the correct successor instruction of a branch").*

□

1.1 Show ob acyclic

Lemma 2. *Let Tr be a valid trace of Flat Operational that has no restarts or discarded instruction tree branches, which induces the relations po, co, rf, and rmw. Let Edge be an edge from ARMv8*

Axiomatic ob*. Then the following property holds for Tr:*

- *Edge: barrier E → barrier E'. In Tr E is committed before E'.*
- *Edge: barrier E → write E'. In Tr E is committed before E' is propagated.*
- *Edge: write E → barrier E'. In Tr E is propagated before E' is committed.*
- *Edge: write E → write E'. In Tr E is propagated before E'.*
- *Edge: barrier E → read R. In Tr E is committed before R is satisfied.*
- *Edge: write E → read R. In Tr E is propagated before R is satisfied.*
- *Edge: read R → barrier E. In Tr R is satisfied before E is committed.*
- *Edge: read R → write E. In Tr R is satisfied before E is propagated.*
- *Edge: read R → read R'. In Tr R is satisfied before R'.*

Proof. By induction on the definition of ob.

Induction step: *Edge* ∈ obs | aob | dob | bob. Then there are multiple cases:

***Edge* ∈ rfe.**

Then *Edge* : $W \to R$ and W and R are from different threads. In Flat Operational for R to read from R, W must be propagated to memory, so the induction statement holds.

***Edge* ∈ coe.**

Then *Edge*: W → W', and W and W' in co. By assumption the coherence order induced by *Tr* is the same as co. By definition for (W,W') in Flat Operational's coherence order W has to propagated to memory before W'. So the induction statement holds.

***Edge* ∈ fre.**

Then *Edge* : $R \to W$, R and W are from different threads, and there is W' with $(W',R) \in$ rf and $(W',W) \in$ co. Now there are two cases:

R is satisfied by forwarding from W'. Then R is satisfied before W' propagates in *Tr* and by the proof for *Edge* ∈ coe the write W' propagates to memory before W. So R is satisfied before W propagates to memory and the induction statement holds.

R satisfied in memory. Then when R is satisfied W' is already propagated and in *Tr* the write W' propagates before W. Since R reads from the last write to the location of R and $(W',R) \in$ rf it must be that W' is in memory and W is not yet in memory. So R satisfies before W is propagated to memory.

***Edge* ∈ rmw.**

Then *Edge* : $R \to W$. Then R and W are a successful load/store exclusive pair and by definition of pop_commit_store_cand the read R must be finished, and therefore satisfied, for W to commit, hence propagate.

***Edge* ∈ [range(rmw)];rfi;[A].**

Then *Edge* : $W \to R$. Then W is a successful store exclusive and R an acquire read.

Since by definition of Flat Operational R cannot read from W by forwarding W must propagate to memory before R can satisfy.

Edge **∈ addr.**

Then either $Edge : R \longrightarrow R'$ or $Edge : R \longrightarrow W$.

***Edge* : *R* → *R′*.** In *Tr* the read R' can only be satisfied if initiated, so after its address is available, so R must be satisfied before R' is.

***Edge* : *R* → *W*.** In *Tr* the write W can only propagate if initiated, so after its address is available, so R must be satisfied before W propagates.

Edge **∈ data.**

Then $Edge : R \longrightarrow W$. W can only propagate after committing, and commit if its data-feeding memory reads are finished, including being satisfied. So R is satisfied before W propagates in *Tr*.

Edge **∈ ctrl;[W].**

Then $Edge : R \longrightarrow W$. W can only propagate when committed, and by pop_commit_store_cand commit when the memory reads feeding into the register reads of conditional branches are finished, including being satisfied. So R is satisfied before W propagates in *Tr*.

Edge **∈ (ctrl | (addr;po)); [ISB];po;[R].**

Then $Edge : R \longrightarrow R'$. Before R' can finish it has to satisfy. By definition of pop_memory_read_request_cand the read R' can only satisfy if all po-earlier `isb` are finished and therefore committed. By pop_commit_barrier_cand any such `isb` can only commit if the memory reads feeding into the conditional branches po-before the `isb` are finished, including being satisfied, and if all po-earlier memory accesses have their address-feeding memory reads finished, including being satisfied. Therefore R must be satisfied before R' can be satisfied in *Tr*.

Edge **∈ addr;po;[W].**

Then $Edge : R \longrightarrow W$. The write W can only propagate when committed, and commit only when the address-feeding memory reads of all po-earlier memory accesses are finished, including being satisfied. Therefore R must be satisfied in *Tr* before W propagates.

Edge **∈ (addr|data);rfi.**

Then $Edge : R \longrightarrow R'$. For R' to satisfy, the write it reads from must have both address and data available, so any memory read feeding into the address or data register reads of this write has to be satisfied. Therefore R must be satisfied before R' can satisfy.

Edge **∈ (ctrl|data);coi.**

Then $Edge : R \longrightarrow W$ and there is W' such that $(R, W') \in$ (ctrl|data) and $(W', W) \in$ coi. By definition of how Flat Operational induces coherence W' must propagate to memory before W does. Before W' propagates it must commit, and therefore all memory reads feeding into the data registers reads of W' and all memory reads feeding into the register reads by conditional branch instructions po-before W' must be finished, including being satisfied. Therefore R must be satisfied before W' can propagate, before W can propagate in *Tr*.

Edge **∈ po;[dmb.full];po.**

Let *DMB* be the dmb. Now there are different cases for the type of the event on the right of the edge. (*RBW* stands for a read, write, or barrier).

Case *Edge* : *RBW* → *R*. R can only satisfy when *DMB* is finished, hence committed. *DMB* can only commit when all po-earlier instructions are finished, including being satisfied (for reads)/committed (for barriers)/propagated (for writes).

Case *Edge* : *RBW* → *B*. B can only commit when *DMB* is finished, hence committed. *DMB* can only commit when all po-earlier instructions are finished, including being satisfied (for reads)/committed (for barriers)/propagated (for writes).

Case *Edge* : *RBW* → *W*. W can only propagate when committed, and it can only commit when *DMB* is finished, hence committed. *DMB* can only commit when all po-earlier instructions are finished, including being satisfied (for reads)/committed (for barriers)/propagated (for writes).

Edge **∈ [L];po;[A].**

Then *Edge* : $W \to R$. By definition of pop_memory_read_request_cand, the acquire read R can only initiate when all po-earlier releases are finished and hence propagated. So W propagated before R satisfied in *Tr*.

Edge **∈ [R];po;[dmb.ld];po;[R|W].**

Let B be the `dmb ld`. Now there are two cases for the type of the event on the right of the edge.

Case *Edge* : *R* → *W*. Then B can only commit after R is finished, including being satisfied, and W can only commit and hence propagate after B is finished and hence committed. So R is satisfied before W propagates in *Tr*.

Case *Edge* : *R* → *R'*. Then B can only commit after R is finished, including being satisfied, and by definition of read_request_cand R' can only satisfy, if B is finished and therefore committed. So R is satisfied before R' in *Tr*.

Edge **∈ [A];po;[R|W].**

Now there are two cases for the type of the event on the right of the edge.

Case *Edge* : *R* → *R'*. Then by definition of read_request_cand R must be completed and hence satisfied before R' can satisfy.

Case *Edge* : *R* → *W*. Then by definition of pop_commit_cand W can only commit and hence propagate after R is finished, including being satisfied in *Tr*.

***Edge* ∈ [W];po;[dmb.st];po;[W].**

Then *Edge* : $W \rightarrow W'$. Let B be the `dmb st`. Then by definition of pop_commit_barrier_cand the barrier B can only commit when W is finished and hence propagated and W' can only commit and hence propagate when B is finished and hence committed. So W propagates before W' in *Tr*.

***Edge* ∈ [R|W];po;[L].**

There are two cases for the type of the event on the right of the edge.

Case *Edge* : $R \rightarrow W$. Then by definition of pop_commit_store_cand the write W can only commit and hence propagate when R is finished and hence satisfied in *Tr*.

Case *Edge* : $W \rightarrow W'$. Then by definition of pop_commit_store_cand the write W' can only commit and hence propagate when W is finished and hence propagated in *Tr*.

***Edge* ∈ [R|W];po;[L];coi.**

Let L be the write release. There are two cases for the type of the event on the right of the edge.

Case *Edge* : $R \rightarrow W$. As shown above L can only commit when R is satisfied, and by definition of pop_write_co_check the write W can only propagate when L is propagated. So R must be satisfied before W can propagate in *Tr*.

Case *Edge* : $W \rightarrow W'$. As shown above L can only commit when W is propagated, and by definition of pop_write_co_check the write W' can only propagate when L is propagated. So W must propagate before W' in *Tr*.

Induction step: *Edge* = *Edge*′; *Edge*″ ∈ ob;ob. This holds by transitivity of implication. □

Corollary 1. *Let Tr be a trace of Flat Operational that has no restarts or discarded instruction tree branches that induces the relations* rf, co, rmw*, and* po*. Then ARMv8-axiomatic's* ob *relation is acyclic for the candidate execution* (po, rf, co, rmw).

Proof. Assume a cycle in ARMv8 Axiomatic;s ob. Let *Edge* be the cycle. So *Edge* : $R \rightarrow R$ or *Edge* : $B \rightarrow B$ or *Edge* : $W \rightarrow W$.

Case *Edge* : $R \rightarrow R$. But from Lemma 2 follows that in *Tr* R is satisfied before R is satisfied. But since *Tr* has no restarts or discarded branches, R can only be satisfied once, contradiction.

Case *Edge* : $B \rightarrow B$. But from Lemma 2 follows that in *Tr* B is committed before B is committed. But every barrier is only committed once, contradiction.

Case *Edge* : $W \rightarrow W$. But from Lemma 2 follows that in *Tr* W is propagated before W is propagated. But every write is only propagated once, contradiction.

□

1.2 Show po-loc | fr | co | rf acyclic

Lemma 3. *Let Tr be a valid trace of Flat Operational that has no restarts or discarded instruction tree branches, which induces the relations po, co, rf, and rmw. Let Edge be an edge from ARMv8 Axiomatic* ob. *Then* po-loc | fr | co | rf *acyclic.*

Proof. Assume a cycle in po-loc | fr | co | rf as induced by the trace *Tr* and let C be the cycle that is derived using a minimal number of po-loc edges, and among the cycles with the same number of po-loc edges minimises the total number of edges. Then C has exactly one po-loc edge.

Assume C has no po-loc edge. Then $C \in$ (rf | co | fr)+. But then $C \in$ (rf | co): assume it is not, so it includes at least one fr edge. (By type rf^{-1};co itself is acyclic.) $C = C'; (E_1, E_2); (E_2, E_3)$ with $(E_2, E_3) \in \mathsf{fr} = \mathsf{rf}^{-1}; \mathsf{co}$. So there exists W such that $(E_2, W) \in \mathsf{rf}^{-1}$ and $(W, E_3) \in \mathsf{co}$. Since E_2 is a read, by type $(E1, E2) \in \mathsf{rf}$, and it is $E_1 = W$. But then $C'; (W, E_3)$ is a shorter cycle with no po-loc edges.

So $C \in$ (rf | co)+. But then by type $C \in$ co+: No edge in (rf | co)+ starts from a read, so rf cannot participate in the cycle C. By definition of Flat Operational's co relation it is $(W, W') \in \mathsf{co}$ if W propagates to memory before W' in *Tr*. But since every write only propagates once, co+ is acyclic.

So assume C has at least one po-loc edge. Show it cannot have multiple. Assume C has multiple po-loc edges. Then it must be $C = C'; (E_1, E_2); C''; (E_3, E_4)$ with $(E_1, E_2) \in$ po-loc and $(E_3, E_4) \in$ po-loc and without loss of generality assuming C'' does not contain a po-loc edge. Since po-loc; po-loc ⊆ po-loc assume also C'' non-empty. So $C'' \subseteq$ (co|rf|fr)+. Now there are different cases for the type of $(E_1, E_2) \in$ po-loc:

$(E_1, E_2) : W \rightarrow W'$. The writes E_1 and E_2 must be coherence related, and it must be $(E_1, E_2) \in$ co; otherwise it is $(E_2, E_1) \in$ co and $(E_2, E_1); (E_1, E_2)$ is a cycle in co; po-loc with fewer po-loc edges. But then C can be constructed without using (E_1, E_2) as po-loc but using (E_1, E_2) as co, contradicting the minimality of C's po-loc edges.

$(E_1, E_2) : W \rightarrow R$. If it is $(E_1, E_2) \in$ rf then (E_1, E_2) can be derived using rf instead of po-loc, contradicting the minimality of po-loc edges in C. So assume (E_1, E_2) not in rf. So $(W', E_2) \in$ rf for some W'. If $(W, W') \in$ co it is $(E_1, E_2) \in$ co; rf; contradicting the minimality of po-loc edges in C. So assume $(W', W) \in$ co. But then it is $(E_2, E_1) \in$ fr and $(E_2, E_1); (E_1, E_2)$ is a cycle in fr; po-loc with a smaller number of po-loc edges than C.

$E_1 \rightarrow E_2 : R \rightarrow W$. Let W' be such that $(W', R) \in$ rf. Now it is either $W' = W$ or not. If $W' = W$ then $(E_2, E_1); (E_1, E_2)$ is a cycle in rf; po-loc with fewer po-loc edges. So assume $W \neq W'$. Then it is either $(W, W') \in$ co or $(W', W) \in$ co. If $(W', W) \in$ co then $(E_1, E_2) \in$ fr and the same cycle can be derived with fewer po-loc edges, contradiction. If $(W, W') \in$ co then $(W', E_1); (E_1, E_2); (E_2, W')$ is a cycle in rf; po-loc; co with fewer po-loc edges (deriving (E_1, E_2) using fr instead of po-loc).

$E_1 \rightarrow E_2 : R \rightarrow R'$. Now there are two cases: R and R' read the same write or not. Assume R and R' read from the same write W. By type $C' = (E5, E6); C'''$ with $(E_5, E_6) \in$ fr. But since R and R' read from the same write it is also $(E_1, E_6) \in$ fr. Contradiction, since then $(E_1, E_6); C'''; (E_3, E_4), C''$ is a cycle with fewer po-loc edges.

So assume $(W, R) \in$ rf and $(W', R') \in$ rf for $W \neq W'$. Now there are two cases: $(W, W') \in$ co or $(W', W) \in$ co. If $(W, W') \in co$ it is $(R, W') \in$ fr, so $(E1_, E_2)$ can be derived using fr; rf contradicting the minimality of po-loc edges in C. So assume $(W', W) \in$ co. But then $(R', W) \in$ fr and $(E2, W); (W, E1); (E1, E2)$ is a cycle in fr; rf; po-loc with fewer po-loc edges, contradiction.

So assume C has exactly one po-loc edge. Then $C \in$ (co | rf | fr)+; po-loc, so $C = C'; P$ for some $C' \in$ (co | rf | fr)+ and $P \in$ po-loc. Now look at all possible cases for the type of P.

Case $P : W \rightarrow W'$. Then $C' : W' \rightarrow W \in$ co+. Assume otherwise. Then by type it must be $C' = C''; (E_1, E_2); (E_2, E_3); C'''$ for some $(E_1, E_2) \in$ rf and $(E_2, E_3) \in$ fr. But then $(E_1, E_3) \in co$ and $C''; (E_1, E_3); C'''; P$ is a shorter cycle in co | rf | fr | po-loc with the same number of po-loc edges. Contradiction to the assumption that C is the shortest cycle.

So $C' \in$ co+. Then by definition of the operational model's coherence it must be that W' propagates to memory before W. But by pop_write_co_check and $(W, W') \in$ po-loc the write W' can only propagate after W is propagated to memory, contradiction.

Case $P : W \rightarrow R$. Then C': $R \rightarrow W \in$ fr. Assume otherwise. Then by type

- Either $C' \in$ fr; co+ = rf^{-1}; co; co+ $\subseteq$ rf^{-1}; co+ $\subseteq$ rf^{-1}; co $\subseteq$ fr, as by definition of Flat Operational's coherence relation co is transitive.
- Or $C' = (E_1, E_2); C''; (E_3, E_4); (E_4, E_5); C'''$ for some $(E_1, E_2) \in$ fr, $(E_3, E_4) \in$ rf, $(E_4, E_5) \in$ fr, and $C'' \in$ co$*$ and $C''' \in$ (co | rf | fr)*. But then $(E_3, E_5) \in$ co and $(E_1, E_2); C''; (E_3, E_5); C'''$ is a shorter cycle, contradicting the assumption of minimality of C.

So $C' : R \rightarrow W \in$ fr. Let W' be the write R reads from. So we have $(W', R) \in$ rf and $(W', W) \in$ co. Now there are two cases: R is satisfied by forwarding or in memory.

- ***R* is satisfied by forwarding.** Then it must be $(W',R) \in$ po-loc because by definition of the satisfy-read-by-forwarding transition W' must be before R in the instruction tree.

  Now assume there is another write po-between W' and R to the address. Let W'' be the po-closest predecessor to R. Then when W'' propagates, by definition of `pop_memory_write_propagate_action_restart_roots` it restarts R since R did not read from W and not from a po-successor of W''. Contradiction to the assumption of *Tr* not having restarts.

  So W' must be the closest po-predecessor of R to the same address. Now there are two cases: $(W, W') \in$ po or $(W', W) \in$ po.

  - **$(W, W') \in$ po.** Then (W, W') and $(W',R) \in$ po, so $(W, W') \in$ po, But then W' by definition of `pop_write_co_check` can only propagate when W is propagated, so it is $(W, W') \in$ co. Contradiction to the assumption $(W', W) \in$ co.
  - **$(W', W) \in$ po.** $(W', W) \in$ po. Since W' is the closest po-predecessor of R to the same address it must be $(R, W) \in$ po. Contradiction to $(W,R) \in$ po-loc.
- ***R* is satisfied in memory.** Then W' is the most recent write to the same address in memory when R is satisfied. By $(W', W) \in$ co the write W' propagates before W. So R is satisfied before W propagates. But when W propagates by definition of `pop_memory_write_propagate_action_restart_roots` the read R is restarted since R is to the address of W and read from a different write but not by forwarding. Contradiction to the assumption that *Tr* has no restarts.

Case $P : R \to W$. Then $C' : W \to R \in$ co?; rf. Proof. Assume otherwise. Then by type C' has to end with an edge from rf, and so $C' = C''; (E_1, E_2); (E_2, E_3); C'''$ with $(E_1, E_2) \in$ rf, $(E_2, E_3) \in$ fr, and $C''' \in$ co?;rf. But then $(E_1, E_3) \in$ co and $C''; (E_1, E_3); C'''; P$ is a shorter cycle. Contradiction to the assumption of the minimality of C.

So there exists W' such that $(W, W') \in$ co? and $(W', R) \in$ rf for some W'. Now there are two cases: $W = W'$ or $W \neq W'$.

- **$W = W'$.** If R is satisfied by forwarding it is $(W, R) \in$ po, contradiction to the assumption. So assume R is satisfied in memory. Then W must be propagated before R is satisfied. But by `pop_write_co_check` the read R must be satisfied before W can propagated, and will not be restarted and satisfied again later, contradiction.
- **$W \neq W'$.** Now there are two cases: R satisfied by forwarding or from storage.
  - ***R* satisfied by forwarding.** Then it is $(W', R) \in$ po-loc and by $(R, W) \in$ po-loc also $(W', W) \in$ po-loc. But then W' can only propagate when W is propagated and it must be $(W', W) \in$ co, contradiction.
  - ***R* satisfied in memory.** Then W' reaches memory before R is satisfied. By as-

sumption of no restarts in *Tr* the read R is only satisfied once. By definition of pop_write_co_check the read R must be satisfied before W propagates. So W' propagates before W and it must be $(W', W) \in$ co, contradiction.

Case $P : R \to R'$. Then $C' : R' \to R \in$ fr; rf. Proof. Assume otherwise. By type C' starts with fr and ends with rf. So it is either (1.) $C' \in$ fr;co+;rf or (2.) $C' \in$ fr;C";rf;fr;C"';rf for some C'' and C'''.

1. But then $C' \in \text{rf}^{-1}$;co;co+;rf $\subseteq \text{rf}^{-1}$;co+;rf $\subseteq \text{rf}^{-1}$;co;rf $\subseteq$ fr;rf.
2. But then $C' \subseteq$ fr;C'';rf;rf^{-1};co;C''';rf $\subseteq$ fr;C'';co;C''';rf, for some C'' and C'''. Contradiction to the assumption of minimality of C.

Let $(W, R) \in$ rf and $(W', R') \in$ rf. Then it is $(W', W) \in$ co by definition of fr. W is either from the same thread as R and R' or not. Assume W is from the same thread as R and R'. Then it must be either $(R, W) \in$ po or $(W, R) \in$ po. If $(R, W) \in$ po then there is a cycle in rf;po-loc using only R and W that has already been dealt with in Case $P : R \to W$. If $(W, R) \in$ po, then it is $(W, R) \in$ po-loc and $(R, R') \in$ po-loc, so also $(W, R') \in$ po-loc and there is a smaller cycle in po-loc;fr using only R' and W that has been dealt with in Case $P : W \to R$.

So assume W is from a different thread than R and R'. Then R must read from W in memory. It cannot be (R, W') in po since otherwise there would be a smaller cycle of R, W', and W in po-loc;co;rf. Now there are two cases: R' is satisfied before R or after.

R' is satisfied before R. If R' is satisfied before R, at the point where R is satisfied by definition of satisfy_read_action_restart_roots the read R' is restarted since it reads from a different write from R that is not po-after R, contradicting the assumption of no restarts in *Tr*. So assume R is satisfied before R'.

R' is satisfied after R. By $(W', W) \in$ co it must be that W' propagates to memory before W does, and since R reads from W in memory, W propagates to memory before R satisfies. So W' propagates, and then W propagates, before R' satisfies. But then R' cannot read from W': it cannot read from W' by thread-local forwarding, since W' is already propagated when R' is satisfied; and it cannot read from W' in memory since W propagated to memory after W' before R' is satisfied, overwriting W'.

Thus in *Tr* by definition of Flat Operational: po-loc | fr | co | rf acyclic. □

1.3 Show rmw & (fre; coe) empty

Lemma 4. *Let Tr be a valid trace of Flat Operational that has no restarts or discarded instruction tree branches, which induces the relations po, co, rf, and rmw. Let Edge be an edge from ARMv8 Axiomatic* ob. *Then* rmw & (fre; coe) *empty.*

Proof. Now assume rmw & (fre; coe) non-empty. So there exists a successful load/store exclusive pair (RE, WE) such that *RE* reads from a write W and there exists a write W' to the same address as *RE* but from a different thread such that $(W, W') \in \mathsf{co}$ and $(W', WE) \in \mathsf{co}$. Then it must be that W propagates to memory before W', and W' propagates to memory before *WE*. Now there are two cases: *RE* is satisfied by thread-internal forwarding or in memory.

***RE* satisfied by forwarding.** Here there are again two cases: (1.) At the point where W propagates to memory *WE* has promised its success or (2.) not.

1. Then when W propagates to memory it adds $t = (RE,[(W,\_)])$ to flat_ss_exclusive_reads and by definition of Flat Operational t can only be removed by the unique write exclusive *WE* paired with *RE*.

   Since W' propagates after W and before *WE*, when W' propagates t is in flat_ss_exclusive_reads. But since *RE* and W' are from a different thread and to the same address W' cannot propagate, contradiction.
2. *WE* must promise its success before propagating. When *WE* promises its success W is already propagated to memory, by assumption. For *WE* to be able promise its success there can be no write coherence-after W (overlapping W) from a different thread than *RE* in memory. So W' cannot be propagated yet and must propagate after the promise-write-exclusive-success transition of *WE*.

   WE's promise-write-exlusive-success transition now adds $(RE,[(W,\_)])$ to flat_ss_exclusive_reads. Since $(RE,[(W,\_)])$ can only be removed from flat_ss_exclusive_reads when propagating *WE*, and since W' propagates before *WE* by assumption, this element must be in flat_ss_exclusive_reads when W' propagates. But since W' overlaps W and is from a different thread than *RE*, the write W' cannot propagate. Contradiction.

***RE* satisfied in memory.** Since *RE* reads from W in memory it has to enter memory after W. *RE*'s satisfy-read transition returns the last memory write written to its location, and since W' propagates after W, the read *RE* satisfies before W' propagates, which in turn is before *WE* propagates. Now there are two cases: 1. when *RE* is satisfied *WE* has promised its success or 2. not.

1. Then when *RE* reads from W in memory it adds $t = (RE,[(W,\_)])$ to flat_ss_exclusive_reads. Since t can only be removed when propagating *WE* and since W' propagates before *WE*, when W' propagates t must be in flat_ss_exclusive_reads. But since W' is to the same address as *RE* and W but from a different thread than *RE*, the write W' cannot propagate, contradiction.
2. *WE* must promise its success before propagating. For *WE* to promise its success after *RE* is satisfied there must be no write overlapping W from a different thread

coherence-after W in memory. So W' cannot be propagated to memory when promising the success of *WE*.

The promise-write-exclusive-success transition for *WE* Flat adds $t = (RE,[(W,\_)])$ to flat_ss_exclusive_reads. Since t can only be removed when propagating *WE* and since W' propagates before *WE*, when W' propagates, t must be in flat_ss_exclusives. But since W' is to the same address as *RE* and W but from a different thread than *RE*, the write W' cannot propagate, contradiction.

Therefore rmw & (fre;coe) empty. □

Theorem 2. *Let Tr be a valid trace of Flat Operational that induces the relations* po*,* rf*,* co*, and* rmw*. Then* $C = (\text{po}, \text{rf}, \text{co}, \text{rmw})$ *is a legal execution in ARMv8-axiomatic*

Proof. Let Tr' be an equivalent Flat Operational trace that induces the same relations po, rf, co, and rmw and has no restarts or discarded instruction tree branches, by Lemma 1. By Corollary 1 C satisfies the external axiom, by Lemma 3 C satisfies the internal axiom, and by Lemma 4 C satisfies the atomic axiom. □

2 Flat-axiomatic definition

In order to show that any behaviour allowed by the ARMv8 Axiomatic model is allowed by Flat Operational, define an intermediate model, called *Flat Axiomatic*. Flat Axiomatic is supposed to express the rules of Flat Operational as precisely as possible as an axiomatic model defined in herd. The model is defined as follows, with the intuition behind some of the definitions given as comments inline.

```
(* Exclude all executions that violate coherence ... *)
acyclic po—loc | rf | fr | co as internal
(* ... and the read/write exclusive guarantee. *)
empty rmw & (fre;coe) as exclusives

let Xw = range(rmw) (* successful store exclusives *)
let Xr = domain(rmw) (* load exclusives *)

(* auxiliary definitions, explained when used in the rules below: *)
let po—R—loc = po—loc \ (po—loc; [W]; po—loc)
let po—no—W—loc = po \ (po; [W]; po—loc)

(* Define the relations in the rmem thread subsystem as relations
between barrier committing or finishing / write propagating or
```

```
finishing / read satisfying or finishing.
   R: read, W: write, B: barrier,
   S, satisfy,
   C: barrier commit / write propagate / read finish *)
let BC_RS = [DMB.SY|ISB|DMB.LD]; po; [R] (* 1 *)
let WC_RS = [Rel];po;[A] (* 2 *)
            | [Xw];rfi;[A] (* 3 *)
            | [W];(po—loc\rf\(po;rf));[R] (* 4 *)
let RS_RS = [A];po;[R] (* 5 *)
            | [R];addr;[R] (* 6 *)
            | [R];(addr|data);rfi;[R] (* 7 *)
            | [R];(po—loc\(rf^—1;rf)\(po;rf));[R] (* 8 *)
let RS_RC = id (* 9 *)
let RC_RC = [R];addr;[R] (* 10 *)
            | [R];addr;po—no—W—loc;[R] (* 11 *)
            | [A];po;[R] (* 12 *)
            | [R];ctrl;[R] (* 13 *)
            | [R];(addr|data);rfi;[R] (* 14 *)
            | [R];po—R—loc;[R] (* 15 *)
let WC_RC = [Rel];po;[A] (* 16 *)
            | [W];(po—R—loc\rf);[R] (* 17 *)
let BC_RC = [DMB.SY|ISB|DMB.LD];po;[R] (* 18 *)
let BC_BC = [DMB.SY];po;[F] (* 19 *)
            | [F];po;[DMB.SY] (* 20 *)
let RC_BC = [R];po;[DMB.SY|DMB.LD] (* 21 *)
            | [R];ctrl;[F] (* 22 *)
            | [R];addr;po;[ISB] (* 23 *)
let WC_BC = [W];po;[DMB.SY|DMB.ST] (* 24 *)
let RC_WC = [R];po;[Rel] (* 25 *)
            | [R];addr;[W] (* 26 *)
            | [R];data;[W] (* 27 *)
            | [R];ctrl;[W] (* 28 *)
            | [R];addr;po;[W] (* 29 *)
            | [A];po;[W] (* 30 *)
            | [R];po—loc;[W] (* 31 *)
```

```
            | [R];rmw;[W] (* 32 *)
let WC_WC = [W];po—loc;[W] (* 33 *)
            | [W];po;[Rel] (* 34 *)
let BC_WC = [F];po;[W] (* 35 *)

(* For writes being finished implies being propagated and committed.
   For barrier being finished implies being committed.

1: BARRIERS: Reads can only satisfy state when
   po—earlier DMB.SYs, ISBs, DMB.LDs are finished.
2: REL/ACQ: Load acquires can only satisfy
   when po—earlier write releases are finished.
3: REL/ACQ: Load acquires cannot be satisfied by forwarding from
   store exclusives.
4: COHERENCE: The propagation of a write restarts all po—later reads
   to the same location that did not read from po—after that write.
5: REL/ACQ: Reads can only satisfy when po—earlier acquire reads are
   satisfied.
6: DATAFLOW: Reads cannot satisfy until their address is known.
7: DATAFLOW: Reads can only satisfy when the write they read from has
   its address and data.
8: COHERENCE: If two reads to the same location, R1 —po—> R2, read
   from different writes where R2’s write is not po—after R1, they
   must satisfy in order, or R1’s satisfaction restarts R2.
9: A read can only finish when it is satisfied.
10: DATAFLOW: Reads can only finish if their dataflow is finished.
11: COHERENCE: Reads can only finish if all instructions up to the
    closest po—predecessor write to the same address have their
    address—feeding reads finished.
12: REL/ACQ: Reads can only finish if all po—earlier acquires are
    finished.
13: COHERENCE: Reads can only finish when the memory reads feeding
    into po—earlier conditional branches are finished.
14: COHERENCE: Reads can only finish when the dataflow of the
    write they read from is finished.
15: COHERENCE: (Aproximation) Reads can only finish when all
    same—location reads in—between them and the earliest
```

po—predecessor write to the same address are finished.

16: REL/ACQ: Acquire reads can only finish if all po—earlier release writes are finished.

17: COHERENCE: Reads that are not satisfied by the nearest po—predecessor write can only finish when that write is propagated.

18: BARRIERS: Reads can only finish if all po—earlier DMB.SYs, ISBs, DMB.LDs are finished.

19: BARRIERS: Barriers can only commit if po—earlier DMB.SYs are finished.

20: BARRIERS: DMB.SYs can only commit if po—earlier barriers are finished.

21: BARRIERS: DMB.SYs and DMB.LDs can only commit if po—earlier reads are finished.

22: COHERENCE: Barriers can only commit if all memory reads feeding into po—earlier conditional branches are finished.

23: BARRIERS: ISBs can only commit if po—earlier address—feeding memory reads are finished.

24: BARRIERS: DMB.SYs and DMB.STs can only commit if po—earlier writes are finished.

25: REL/ACQ: Release writes can only commit if po—earlier reads are finished.

26: COHERENCE: Writes can only commit if their data flow is finished (here feeding into the address).

27: COHERENCE: Writes can only commit if their data flow is finished (here feeding into the data).

28: COHERENCE: Writes can only commit if the memory reads feeding into po—earlier conditional branches are finished.

29: COHERENCE: Writes can only commit if po—earlier memory accesses have their address—feeding memory reads finished.

30: REL/ACQ: Writes can only commit if po—earlier read acquires are finished.

31: COHERENCE: (Approximation) Writes can only propagate if po—predecessor same—address reads are finished.

32: EXCLUSIVES: Exclusive writes can only commit after their matching exclusive reads are finished.

33: COHERENCE: No write subsumption: writes can only propagate when all po—previous writes to the same location are propagated.
34: REL/ACQ: Release writes can only commit when po—previous writes are finished.
35: BARRIERS: A write can only commit when po—earlier barriers are finished.

*)

(* Now compose these edges in the relation "order", where
order(E,E’) <=>
E finishes (if E is a barrier) /
propagates or finishes, (E write) /
satisfies (E read)
before
E’ commits (E’ barrier) /
commits or propagates (E’ write) /
satisfies (E’ read).

Edges XX_RS; RC_YY are composable, edges XX_RC; RS_YY are not. To list all the edges that are composable — remove all edges XX_RC and replace them with edges of type XX_RS, XX_BC, or XX_WC;

So:
— delete RC_RC
— replace RS_RC by RS_RC?; RC_RC*; (RC_BC | RC_WC)
— replace WC_RC by WC_RC?; RC_RC*; (RC_BC | RC_WC)
— replace BC_RC by BC_RC?; RC_RC*; (RC_BC | RC_WC)

The resulting order is below.

— Coherence in operational flat axiomatic is determined by the order in which writes propagate to memory. Conversely, for the candidate execution to be allowed by the operational model this write propagation order has to be compatible with the other constraints on the thread behaviuor. Therefore include co in the order.

— If a read reads in storage it reads the most recent write that went into memory before it. Conversely, to get an operational trace where the reads—from is as in the candidate execution for (w,r) in rf the write w has to propagate before r is satisfied and

```
      any write w’ with (w,w’) co —— so (r,w’) in fr —— must not propagate
      until after r is satisfied. If a read reads by forwarding it
      must do this before that write propagates and therefore also before
      any coherence later writes propagate. So include fr in the order.

    — For a read to read from a different—thread write it has to go
      into memory after it. So include rfe in the order.

    To match a valid Flat trace this order has to be acyclic.

  *)

let Order = BC_RS
            | WC_RS
            | RS_RS
            | RS_RC; RC_RC*; (RC_BC | RC_WC)
            | WC_RC; RC_RC*; (RC_BC | RC_WC)
            | BC_RC; RC_RC*; (RC_BC | RC_WC)
            | BC_BC
            | RC_BC
            | WC_BC
            | RC_WC
            | WC_WC
            | BC_WC

            | co
            | rfe
            | fr

Acyclic (Order+) as external
```

3 Flat Axiomatic behaviour included in Flat Operational

3.1 Auxiliary result for load/store exclusives

Lemma 5. *Let* (po, rf, co, rmw) *be a candidate execution of Flat-axiomatic. Let S be a linearisation of* Order. *Let* $(R, W) \in$ rmw *with R and W to the same address, WR the write such that* $(WR, R) \in$ rf. *Then*

1. $(WR, W) \in S$,
2. $(R, W) \in S$,

3. *and there is no W' with $(WR, W') \in S$, $(W', W) \in S$ for a write W' to the same address as W but from a different thread than W.*

Proof. It is $(R, W) \in$ rmw $\subseteq$ RC_WC $\subseteq S$, so (2.) holds. Now either $(WR, R) \in$ rfi, then $(WR, R) \in$ po by internal axiom and by $(R, W) \in$ po also have $(WR, W) \in$ po. Therefore it is $(WR, W) \in$ [W];po-loc;[W] $\subseteq$ WC_WC $\subseteq S$, so have (1.). Or $(WR, R) \in$ rfe $\subseteq S$ and by $(R, W) \in rmw \subseteq RC\_WC \subseteq S$ it is $(WR, W) \in S$, so have (1.).

Since co totally orders same-address writes and co $\subseteq S$ it is $(WR, W) \in$ co by (1.). (3.) Now assume there is such a write W'. Then since co $\subseteq S$ and co totally orders same-address writes, $(WR, W') \in$ co and $(W', W) \in$ co. But then $(R, W') \in$ fre and $(W', W) \in$ coe so that $(R, W') \in$ (fre;coe) & rmw. Contradiction to the Exclusives axiom. □

Lemma 6. *Let* (po, rf, co, rmw) *be a candidate execution of Flat-axiomatic. Let* $(R, W) \in$ rmw *and* $(R', W') \in$ rmw *and R, R', W, W' all from the same thread to the same address, with $R \neq R'$. Let* $(RW, R) \in$ rf *and* $(RW', R') \in$ rf. *Then it cannot be $RW = RW'$.*

Proof. Assume $RW = RW'$. Then it is $(R, W) \in$ po, and $(R', W') \in$ po. Assume without loss of generality $(R, R') \in$ po. Now because of $(R, W) \in$ rmw it cannot be $(R, R'); (R', W) \in$ po. So po must order them as $R;W;R';W'$. But then it must be $(RW, W) \in$ co, so $(R', W) \in$ fr. Contradiction to internal axiom: cycle in fr|po. □

3.2 Write-finish lemma

Lemma 7. *Let St be a state of Flat operational. For any store W: if the write of W is committed, and all eager transitions taken, then W is finished.*

Proof. Since W by assumption is aligned, W is its only write. Hence after committing, W was eagerly completed. Since all register reads done by a store are reads to determine its address and data, such register reads have already been completed, and the pseudocode execution of W can be eagerly finished. Since W has committed its write its data must be fully determined and all program-order preceding conditional branches are finished. Thus the condition for finishing W holds and W has been eagerly finished. □

3.3 Read-finish lemma

Lemma 8. *Let St be a state of Flat operational. For any read R: if R is satisfied, all writes and barriers E with* $(E, R) \in$ (WC_RC | BC_RC); RC_RC* *are finished, and all reads R' with* $(R', R) \in$ RC_RC+ *are satisfied, and all eager transitions taken, then R is finished.*

Proof. By induction on the instruction tree.

Induction start: empty instruction tree. If the instruction tree is empty there is no such read R in it.

Induction hypothesis: assume the statement holds for some instruction tree *IT*, show it also holds for adding a leaf *II* to *IT*. By the induction assumption the induction hypothesis holds for all satisfied reads R in IT also with II added. (The pop_finish_load_cand condition is unaffected by po-later instructions.) So remains to show that the induction hypothesis holds for II. Let R be the leaf II and assume R is satisfied, all writes and barriers E with $(E,R) \in$ (WC_RC | BC_RC); RC_RC* are finished and all reads R' with $(R',R) \in$ RC_RC+ are satisfied. Have to show that R is finished.

For R to finish the following have to hold:

1. commitDataflow. The memory reads feeding into the register reads of R have to be finished. Let R' be one such read. Have $(R',R) \in$ [R];addr;[R]⊆RC_RC. Therefore R' is satisfied in St, all reads $(R'',R') \in$ RC_RC+ are satisfied since $(R'',R');(R',R) \in$ RC_RC+; RC_RC ⊆ RC_RC+, and all writes E' are propagated and barriers E' committed and thus E' eagerly finished for $(E',R') \in$ (WC_RC | BC_RC); RC_RC*, since have $(E',R');(R',R) \in$ (WC_RC|BC_RC); RC_RC*; RC_RC ⊆ (WC_RC|BC_RC); RC_RC*. Then by induction hypothesis R' is finished.
2. commitControlflow. Conditional branches po-before R have to be finished. Assume there is an uncommitted branch instruction po-before R, let BR be the po-earliest one. BR's finish transition is taken eagerly, so if BR is not finished, then it is because BR cannot finish yet.

   Since BR is the po-earliest unfinished branch its control flow is finished. So it must be the dataflow going into BR that is unfinished: there is at least one read R' that feeds into the register reads of BR that is unfinished. But then $(R',R) \in$ [R];ctrl;[R] ⊆ RC_RC and therefore R' satisfied and all writes E' are propagated and barriers E' committed and E' therefore eagerly finished for $(E',R') \in$ (WC_RC|BC_RC); RC_RC*, since $(E',R');(R',R) \in$ (WC_RC|BC_RC);RC_RC*, and all reads R'' with $(R'',R') \in$ RC_RC+ satisfied, since $(R'',R');(R',R) \in$ RC_RC+. Therefore by induction hypothesis R' is finished, contradiction.
3. All po-earlier `dmb sy`, `dmb ld`, `isb` are finished. Since [DMB.SY|ISB|DMB.LD];po;[R] ⊆ BC_RC, by assumption all of these are finished in St.
4. All po-earlier acquire reads are finished.

   Let R' be a po-earlier acquire. So $(R',R) \in$ [Acq];po;[R] ⊆ RC_RC. Then R' satisfied. And all writes E' are propagated and barriers E' committed and E' thus eagerly finished for $(E',R') \in$ (WC_RC|BC_RC);RC_RC*, by $(E',R');(R',R) \in$ (WC_RC|BC_RC);RC_RC*, and all reads R'' with $(R'',R') \in$ RC_RC+ are satisfied, since $(R'',R');(R',R) \in$ RC_RC+. Then by induction hypothesis R' is finished.

5. If R is an acquire then all po-earlier releases are finished. This is true in St, since by [Rel];po;[Acq] $\subseteq$ WC_RC these write releases are propagated and therefore eagerly finished.
6. If the closest po-earlier write W to the same address was forwarded to R the memory reads feeding into the register reads of W must be finished.

   Let R' be one such read. Then $(R',R) \in$ [R];(addr|data);rfi $\subseteq$ RC_RC. Then by assumption R' is satisfied, all reads R'' with $(R'',R') \in$ RC_RC+ are satisfied, since it is $(R'',R');(R',R) \in$ RC_RC+, and all writes E' are propagated and barriers E' committed and therefore E' eagerly finished for $(E',R') \in$ (WC_RC | BC_RC); RC_RC*, since $(E',R');(R',R) \in$ (WC_RC | BC_RC); RC_RC*. Then by induction hypothesis R' is finished.
7. If the closest po-earlier write W to the same address was not forwarded to R it must be propagated. By assumption R is satisfied in St. Now there are two cases:

   $(W,R) \in$ rf. Since by assumption W was not forwarded to R, the read R read from W in memory, so W propagated.

   $(W,R) \notin$ rf. Then $(W,R) \in$ [W];(po-R-loc\rf);[R] $\subseteq$ WC_RC, so by assumption W is finished and therefore propagated.
8. All memory accesses between this write W and R have their address-feeding memory reads finished and are initiated.

   Let R' be one such read. Then $(R',R) \in$ [R];addr;po-no-W-loc;[R] $\subseteq$ RC_RC $\subseteq S$. Therefore R' is satisfied, all reads R'' with $(R'',R') \in$ RC_RC+ are satisfied, since $(R'',R');(R',R) \in$ RC_RC+, and all writes E' are propagated and barriers E' committed and therefore E' eagerly finished for $(E',R') \in$ (WC_RC | BC_RC); RC_RC*, since $(E',R');(R',R) \in$ (WC_RC | BC_RC); RC_RC*. Then by induction hypothesis R' is finished. Moreover, since all such R' are satisfied, all memory accesses between W and R have eagerly done the register reads necessary to determine their address and eagerly initiated.
9. All reads R' to the same address between W and R must be satisfied and not-restartable. Let R' be such a read. Then $(R',R) \in$ [R];po-R-loc;[R] $\subseteq$ RC_RC. Therefore R' is satisfied, all reads R'' with $(R'',R') \in$ RC_RC+ are satisfied, since $(R'',R');(R',R) \in$ RC_RC+, and all writes E' are propagated and barriers E' committed and therefore E' finished for $(E',R') \in$ (WC_RC|BC_RC); RC_RC*, because it is $(E',R');(R',R) \in$ (WC_RC|BC_RC); RC_RC*. Then by induction hypothesis R' is finished.

Now have: R is satisfied, so the load is eagerly completed. Since there are no additional register reads to be done by R its pseudocode execution is finished. Since all the conditions for finishing R hold in St and the read-finish transition is an eager transition, R is finished and the induction hypothesis holds. $\square$

3.4 Main proof

Theorem 3. *If a candidate execution* $(\mathsf{po},\mathsf{rf},\mathsf{co},\mathsf{rmw})$ *is allowed in Flat-axiomatic, then there exists a trace Tr of Flat Operational where for each* $(W,R)\in\mathsf{rf}$ *the read R is satisfied by the write W, where for each* $(W,W')\in\mathsf{co}$ *the write W is propagated before W′, where for each* $(R,W)\in\mathsf{rmw}$ *the read R and the write W are a successful read-write-exclusive pair, and where the instruction tree viewed as a relation is* po.

Proof. Let S be a linearisation of the candidate execution's Order. Define $\mathsf{rfs} = \mathsf{Order}\,\&\,\mathsf{rf}$, $\mathsf{rft} = \mathsf{rf}\setminus\mathsf{rfs}$, $Rs = \mathsf{range}\,\mathsf{rfs}$, $Rt = \mathsf{range}\,\mathsf{rft}$. Then $\mathsf{rfe}\subseteq\mathsf{rfs}$ since $\mathsf{rfe}\subseteq\mathsf{Order}$.

3.4.1 Trace construction

Now construct *Tr* as follows:

1. Start with an empty trace.
2. Fetch all instructions one-by-one following po of the candidate-execution. In program order, for each write exclusive W:
   - if $W\in\mathsf{range}\,\mathsf{rmw}$, promise the success of the write exclusive
   - if $W\notin\mathsf{range}\,\mathsf{rmw}$ promise the failure of the write exclusive
3. Take all enabled eager transitions.
4. For each next element E of S:
   - 4.1. **If E is a read $R\in Rt$** satisfy R by forwarding from the unique W with $(W,R)\in\mathsf{rf}$ using the transition $T = \mathsf{T\_mem\_forward\_write}$.

     If E is a read $R\in Rs$ satisfy R in memory with $T = \mathsf{T\_Flat\_mem\_satisfy\_read}$ with the write W with $(W,R)\in\mathsf{rf}$.

     If E is a write W take transition $T = \mathsf{T\_propagate\_mem\_write}\,W$.

     If B is a barrier take transition $T = \mathsf{T\_commit\_barrier}\,B$.
   - 4.2. Take all enabled eager non-fetch, non-barrier-commit transitions (these include the pseudocode-internal transitions, register reads and writes, initiating of reads and writes, completing loads and stores that have done all their respective reads and writes, and finishing instructions).

3.4.2 Trace is valid trace

Now show by induction on n: if *Tr* is the trace constructed for $S[0..n]$, then:

0. Assume $(RE,[(W,\_)]) \in \mathsf{flat\_ss\_exclusive\_reads}$ after executing *Tr*. Then $(W,RE) \in \mathsf{rf}$ and there exists a write exclusive *WE* such that $(RE,WE) \in \mathsf{rmw}$. Let *WE* be this write exclusive. Then $W \in S[0..n]$ and $WE \notin S[0..n]$ and *RE* satisfied after *Tr*.
1. For all satisfy-read transitions in *Tr* for reads *R* (whether by forwarding or from memory), the write *W* it reads from is the write in (W,R) in the rf relation from the candidate execution above. For all write-propagate transitions in *Tr* for writes *W*, they occur in *Tr* consistent with the co relation in the candidate execution above. For each $(R,W) \in \mathsf{rmw}$ in the candidate execution above, the write exclusive *W* is successfully paired with $R \in Tr$. The instruction tree viewed as a relation is po.
2. *Tr* involves no restarts and no discarding of branches in the instruction tree.
3. *Tr* is a valid (partial) trace of Flat Operational.
4. If there are any enabled transitions after *Tr*, they can only be one of:
    - satisfy memory read by forwarding
    - satisfy memory read from storage
    - propagate memory write
    - commit barrier
    - fetch next instruction

Induction start, empty prefix of S.

0. After executing only fetch and promise-write-exclusive-success or promise-write-exclusive-failure transitions the flat_ss_exclusive_reads component of the state is empty.
1. Since the fetch and promise-write-exclusive-success/failure transitions do not satisfy reads or commit writes (1.) holds for rf and co. Since by construction of *Tr* the success of a write exclusive *W* is promised if-and-only-if $W \in \mathsf{range}\,\mathsf{rmw}$, (1.) holds for rmw as well. By construction the instruction-tree unfolding matches po, since by proof of (2.) the instruction tree is not pruned.
2. By definition of Flat Operational fetching and the promise-write-exclusive-success transition cannot cause restarts. The promise-write-exclusive-failure transition of a write *WE* can only restart reads that have read from *WE*. But by construction no reads are satisfied yet. Neither fetching nor the promise-write-exclusive-success/failure transition discard instruction tree branches.
3. By definition of Flat Operational for "constant" conditional branches both possible targets of the branch can be fetched, for computed branches any address can be fetched.
4. By definition, all the eager transitions (all except the ones in the inductive hypotheses) are taken.

    For each $(R,W) \in \mathsf{rmw}$ it is $(R,W) \in \mathsf{po}$ with no other exclusive instruction in-between

R and W in program order. So the promise-write-exclusive-success transition for W is enabled. For all unsuccessful write exclusive the promise-write-exclusive-failure transition is enabled.

Induction step: n → n+1. Now extend Tr to Tr' for the next element $E = S[n+1]$.

Case E is a read R from Rt. Show extending the trace for E preserves properties 0. – 3..

0. Only the promise-write-success, satisfy-read-in-memory and propagate-memory-write transitions change the flat_ss_exclusive_reads field. Since E is a read in Rt and satisfied by forwarding, 0. still holds by induction hypothesis.
1. By induction hypothesis the trace Tr induces the rf, co, and rmw relations from the candidate execution, the instruction tree matches po, extending Tr to Tr' preserves this. Only have to show that extending Tr to Tr' for E preserves this as well. But this follows from the construction of Tr': R is satisfied by W for $(W,R) \in \mathsf{rf}$. The relations po, co and rmw are unaffected.
2. Have to show that the satisfaction of R does not cause the restart or discarding of any instructions. By definition T does not discard instruction tree branches. Let restart_roots be the result of calling pop_memory_read_action_restart_roots. These and their dependent reads or writes are the instructions that will be restarted as part of the memory read action. Then restart_roots is the set of all reads R' with $(R,R') \in \mathsf{po\text{-}loc}$ for which R' has been satisfied, but by neither W nor a write po-between R and R' after executing Tr'. Show that restart_roots for this memory read action T is empty. From that follows that extending the trace for E preserves 2..

   Assume there is such a read R' that is satisfied by a write W' that is neither W nor po-between R and R' after executing Tr'. Then have $(R,R') \in \mathsf{po\text{-}loc}$, and therefore $(R,R') \in \mathsf{[R];(po\text{-}loc\setminus(rf^{-1};rf)\setminus(po;rf));[R]} \subseteq \mathsf{RS\_RS} \subseteq S$. By construction R' can only have already been satisfied if $(R',R) \in S$. But already have $(R,R') \in S$. Contradiction to the acyclicity of S.
3. To be able satisfy R from W by thread-internal forwarding,
   - 3.1. R must be in MOS_pending_mem_read state and read_request_cand hold,
   - 3.2. W must be in state MOS_potential_mem_writeq with data available,
   - 3.3. W must be before R in the instruction tree,
   - 3.4. there must be no write W' to the same address between R and W in the instruction tree,
   - 3.5. there must be no R' to the same address in between R and W in the instruction tree that read from another write $W \neq W'$,

3.6. if R is a read acquire, then W is not a store exclusive.

3.1. After executing *Tr* the read R is in MOS_pending_mem_read state and read_request_cand holds.

The reads feeding into the register reads of R have been satisfied and therefore eagerly completed: for any read R' whose read value feeds into the address of R it is $(R',R) \in \mathsf{RS\_RS} \subseteq \mathsf{Order} \subseteq S$; therefore by construction R' is satisfied and eagerly completed after executing *Tr*; the register writes of any write exclusive *WE*'s success bits feeding into the address of R are eagerly done, since by construction *WE* has been promised to be successful; any other, non-memory, instructions feeding into R's registers reads have been done eagerly.

Remains to show that read_request_cand holds and that R is not already satisfied after *Tr*. Then, since register read transitions and load initiation are eager, 3.1. follows.

Show the predicate pop_memory_read_request_cand holds. This is true if

3.1.1. All program-order earlier `dmb sy`, `isb`, `dmb ld` are finished. By definition of BC_RS have [DMB.SY|ISB|DMB.LD];po;[R] $\subseteq \mathsf{BC\_RS} \subseteq S$. So by construction of *Tr* all `dmb sy`, `isb`, `dmb ld` are committed after executing *Tr*, and therefore eagerly finished after *Tr*.

3.1.2. If R is an acquire read then all po-earlier write releases are finished. It is [Rel];po;[Acq] $\in$ WC_RS $\subseteq$S. So by construction all po-earlier write releases are propagated if R is an acquire, and therefore eagerly completed and finished after *Tr*.

3.1.3. All po-earlier acquire reads are completed. It is [Acq];po;[R] $\subseteq \mathsf{RS\_RS} \subseteq S$. So by construction all acquires po-before R are satisfied and eagerly completed after executing *Tr*.

R cannot be satisfied after *Tr*: by construction R could only be satisfied after *Tr* if $(R,R) \in S$, but S is acyclic.

3.2. After executing *Tr* the write W is in state MOS_potential_mem_write state with data available.

The memory reads feeding into the register reads of W have been completed: for any read R' that feeds into W's address or data it is $(R',R) \in \mathsf{RS\_RS} \subseteq \mathsf{Order} \subseteq S$; therefore by construction R' is satisfied after executing *Tr*$'$ and therefore eagerly completed; any register write of a write exclusive's success bit feeding into the address or data of W has been done eagerly, since all write exclusives have already promised success or failure after *Tr*; all non-memory instructions feeding into W' registers reads have been done eagerly.

To show W is in state MOS_potential_mem_write after executing *Tr*, remains to show that W has not propagated yet.
By construction W can only be propagated if $(W,R) \in S$, so assume $(W,R) \in S$. But by definition of rft it is $(R,W) \in S$. Since S acyclic, contradiction: $(W,W) \in S$.

3.3. This follows by definition of the operational model if $(W,R) \in \mathsf{po}$. As shown above, rfe $\subseteq rfs$. Therefore, since rft $=$ rf $\setminus$ rfs it is $(W,R) \in$ rfi, so W and R from the same thread. And it must be $(W,R) \in \mathsf{po}$ because the coherence axiom requires the acyclicity of po | rf.

3.4. This follows by definition of the operational model if there is no $(W,W') \in$ po and $(W',R) \in$ po for a W' to the same address. There cannot be such $(W,W') \in$ po and $(W',R) \in$ po since by the coherence axiom in the candidate execution po | co | rf | fr acyclic.

3.5. Assume there is such a read R'.
Then by construction $(R',R) \in S$, and by induction hypothesis $(W',R') \in$ rf of the candidate execution. Now in the candidate execution W and W' must be coherence related. If it is $(W',W) \in$ co, then $(R',W');(W',W) \in \mathsf{rf}^{-1}$;co $=$ fr and there is a cycle in fr; po, contradiction to the assumption that Flat-axiomatic's internal axiom holds for the candidate execution.
So assume it is $(W,W') \in$ co. By assumption it is $(W,R) \in$ rf, so $(R,W') \in$ fr of the candidate execution. But then there is a cycle in fr; rf; po in the candidate execution. Contradiction to the assumption that the internal axiom holds.

3.6. Assume R is a read acquire and W a store exclusive. But then by definition of rfs the read R is required to be in *Rs*, since [Xw];rfi;[A] $\subseteq$ WC_RS $\subseteq$ Order $\subseteq S$, contradiction.

Case E is a read R from *Rs*. Show extending the trace for E preserves properties 0. – 3..

0. The transition T for R only changes the flat_ss_exclusive_reads field in case it is a read exclusive *RE* with a program-order following write exclusive *WE* for which *Tr* contains the promise-write-exclusive-success transition. By construction in that case $(RE, WE) \in$ rmw. Let W be the write with $(W,RE) \in$ rf. Then T adds the element $(RE,[(W,\_)])$ to flat_ss_write_exclusives. It is $(W,RE) \in$ rfs $\subseteq S$, so $W \in S[0..n+1]$. By construction, after Tr' the read $R = RE$ is satisfied. Left to show that $WE \notin S[0..n+1]$. This follows by $(RE, WE) \in$ fr $\subseteq S$. Since R is a read, for the other elements in flat_ss_exclusive_reads 0. still holds.
1. By induction hypothesis the trace *Tr* induces the rf and co relation from the candidate execution, the instruction tree matches po. Have to show that extending

Tr to Tr' for E preserves this. But this follows from the construction: R is satisfied by W for $(W,R) \in$ rf. po, co, and rmw are unchanged.

2. Have to show that the satisfaction of R does not cause the restart or discarding of any instructions. By definition T does not discard instruction-tree branches. Let restart_roots be the result of calling pop_memory_read_action_restart_roots. These and their dependent reads or writes are the instructions that will be restarted as part of the memory read action. Then restart_roots is the set of all reads R' with $(R,R') \in$ po-loc for which R' has read neither from W nor from a write po-between R and R'.
   Show that after executing *Tr* restart_roots for this memory read action T is empty. From that follows that extending the trace to Tr' for E preserves 2..
   Assume there is such a read R' that is satisfied by a write W' that is neither W nor po-between R and R' after executing Tr'.
   Then $(R,R') \in$ [R];(po-loc \ (rf^{-1};rf) \ (po;rf));[R] $\subseteq$ RS_RS $\subseteq S$. But by construction R' can only have been satisfied if it was already satisfied, so if $(R',R) \in S$. Contradiction to the acyclicity of S.
3. To be able to satisfy R from W in memory after Tr',
   - 3.1. R must be in MOS_pending_mem_read state and read_request_cand hold,
   - 3.2. W must be propagated to memory.
   - 3.3. There must not be a write W' to the same address that propagated after W
   - 3.4. If R is an exclusive read in the storage subsystem state there must be no element (*RE*',[(W',_)]) $\in$ flat_ss_exclusive_reads where *RE*' and R are to the same address but from different threads.
   - 3.1. After executing *Tr* the read R is in state MOS_pending_mem_read state and read_request_cand holds.
     The memory reads feeding into the register reads of R have been completed: for any read R' whose read value feeds into the address of R it is $(R',R) \in$ RS_RS $\subseteq$ Order $\subseteq S$; therefore by construction R' is satisfied and therefore eagerly completed after executing *Tr*; the register writes of the success bit of any store exclusive feeding into the register reads of R have been done eagerly since all write exclusives have promised success or failure after *Tr*; all non-memory data dependent instructions feeding into the register reads of R have been done eagerly.
     Remains to show pop_memory_read_request_cand holds and that R is not already satisfied after *Tr*. Since register read transitions and load initiation are eager, then 3.1. follows.

Show pop_memory_read_request_cand holds. This is true if

3.1.1. All program-order earlier `dmb sy`, `isb`, `dmb ld` are finished. By definition of BC_RS it is [DMB.SY|ISB|DMB.LD];po;[R] $\subseteq$ BC_RS $\subseteq S$. So by construction of *Tr* all `dmb sy`, `isb`, `dmb ld` are committed after executing *Tr*, and therefore eagerly finished.

3.1.2. If R is an acquire read then all po-earlier write releases are finished. It is [Rel];po;[Acq] $\in$ WC_RS $\subseteq S$. So by construction all po-earlier write releases are propagated if R is an acquire, and therefore eagerly completed and finished.

3.1.3. All po-earlier acquires are completed. It is [Acq];po;[R] $\in$ RS_RS $\subseteq S$. So by construction all acquires po-before R are satisfied and eagerly completed after executing *Tr*.

R cannot be satisfied after *Tr*: by construction R could only have been satisfied if there was $(R,R) \in S$. But S is acyclic.

3.2. After executing *Tr* the write W is propagated. By definition of rfs it is $(W,R) \in$ rfs $\subseteq S$, so by construction W has already been propagated.

3.3. Assume after executing *Tr* there is a write W' to the same address that propagated after W. Then this must be because $(W,W') \in S$ and $(W',R) \in S$. Since co $\subseteq S$ and co totally orders same-address writes it must be $(W,W') \in$ co and therefore $(R,W') \in$ fr $\subseteq S$. But then S cyclic. Contradiction.

3.4. Assume R is a read exclusive $R = RE$ and $(RE,WE) \in$ rmw. Assume there is such a (*RE*',[(W',_)]) $\in$ flat_ss_exclusive_reads. Then by induction hypothesis it is (W',*RE*') $\in$ rf, there is *WE*' such that (*RE*',*WE*') $\in$ rmw, $W' \in S[0..n]$, *RE*' satisfied after Tr', and *WE*' not in $S[0..n]$. Then have $(W',RE) \in S$. Also have $(W,RE) \in$ rfs $\subseteq S$. Since *RE*' is satisfied, by construction it must be *RE*' $\in S[0..n]$ and therefore (*RE*',*RE*) $\in S$. And it is (*RE*,*WE*') $\in S$.

Also have $(RE,WE) \in$ fr $\subseteq S$. Since by assumption *RE* and *RE*' from different threads it must also be that *WE* and *WE*' are from different threads.

Now there are two cases:

$(WE,WE') \in S$. Then it is $(W',RE) \in S$, $(RE,WE) \in S$, and $(WE,WE') \in S$. But *WE* and *WE*' are to the same address from different threads, contradicting Lemma 5 for (R',*WE*') $\in$ rmw.

$(WE',WE) \in S$. Then it is $(W,RE) \in S$, (*RE*,*WE*') $\in S$, and $(WE',WE) \in S$. But *WE* and *WE*' are to the same address from different threads, contradicting Lemma 5 for $(R,WE) \in$ rmw.

Case E is a write W. Show extending the trace for E preserves properties 0. – 3..

0. Assume $t = (RE', [(W', \_)]) \in \mathsf{flat\_ss\_exclusive\_reads}$ after executing Tr'. Now there are two possibilities:

   $t \in$ flat_ss_exclusive_reads before T. Then by IH there is $(W', RE') \in \mathsf{rf}$ and there exists WE' such that $(RE', WE') \in \mathsf{rmw}$ with $W' \in S[0..n]$ and WE' not in $S[0..n]$ and RE' satisfied after Tr'. Have to show T preserves this. rf and rmw are unaffected by T, after T it will still be W' in $S[0..n+1]$ and RE' still satisfied. So have to show WE' not in $S[0..n+1]$. Assume it is. Then $W = WE'$. But then $T = \mathsf{T\_propagate\_write}\, W$ is annotated with RE' and $(RE', [(W', \_)])$ deleted by T from flat_ss_exclusive_reads, contradiction.

   otherwise. Then RE' is a read exclusive that was satisfied by thread-internal forwarding from W and it is paired with a write exclusive WE' for which Tr contains the promise-write-success transition. Then have $(W, RE') \in \mathsf{rf}$. By construction it is $(RE', WE') \in \mathsf{rmw}$. RE' is satisfied and it is W in $S[0..n+1]$. Still have to show that WE' not in $S[0..n+1]$.

   By $(RE', WE') \in \mathsf{rmw}$ it is $(RE', WE') \in \mathsf{po}$. Since RE' read from W by thread-internal forwarding it is also $(W, RE') \in \mathsf{po}$, so $(W, WE') \in \mathsf{po}$ and they are writes to the same location. But then it is $(W, WE') \in \mathsf{[W];po\text{-}loc;[W]} \subseteq \mathsf{WC\_WC} \subseteq S$, and since $W = S[n+1]$ it must be that $WE' \notin S[0..n+1]$.
1. By induction hypothesis the trace Tr induces the rf, co, and rmw relation from the candidate execution, the instruction tree matches po. Have to show that extending Tr' to Tr'' for E preserves this. Since in the operational model the coherence relation is determined by the order in which writes reach memory, have to show: (1.1.) for all $(W', W) \in \mathsf{co}$ the write W' has already been propagated after Tr and (1.2.) for all W' to the same address as W that have been propagated before W in Tr it is $(W', W) \in \mathsf{co}$.
   1.1. Since $\mathsf{co} \subseteq S$ by construction of Tr all such writes W' have already been propagated.
   1.2. Let W' be any same-address write that is propagated after in Tr. Then by construction of Tr it must be $(W', W) \in S$. But since $\mathsf{co} \subseteq S$, since co totally orders all same-address writes, and since S is acyclic W' it must be $(W', W) \in \mathsf{co}$.

   po, rf, and rmw are unaffected by T, so 1. follows.
2. Have to show that the propagation of W does not cause the restart or discarding of any instructions. By definition T does not discard instruction-tree branches. Let restart_roots be the result of calling propagate_write_action_restart_roots. These and their dependent reads or writes are the instructions that will be restarted as

part of the memory write propagate action. Then restart_roots is the set of all reads R with $(W,R) \in$ po-loc for which R has been satisfied, but neither from W nor from a write po-between W and R after *Tr*. Show that after executing *Tr* restart_roots for this memory write action is empty. From that follows that extending the trace for E preserves 2..

Assume after executing *Tr*, R is such a read that is satisfied by write W' that is neither W nor po-between W and R. So $(W,R) \in$ [W];(po-loc\rf\(po;rf));[R] $\subseteq$ WC_RS $\subseteq S$. By construction of *Tr* the read R can only have already been satisfied if $(R,W) \in S$. Contradiction to the acyclicity of S.

3. To propagate W, its store has to be committed and pop_write_co_check has to hold. Since write commitment transitions are eager only have to show:

   3.1. W must be in state MOS_potential_mem_write with its data available

   3.2. pop_commit_store_cand and pop_write_co_check must hold.

   3.3. There exists no (*RE*',[(W',_)]) $\in$ flat_ss_exclusive_reads after *Tr*$'$ where *RE*' and W are to the same address but from different threads.

   3.1. After executing *Tr* the write W is in state MOS_potential_mem_write.
   The memory reads feeding into the register reads of W have been satisfied: for any read R whose read value feeds into the address and data of W it is $(R,W) \in$ RC_WC $\subseteq$ Order $\subseteq S$; therefore by construction of *Tr* the read R is satisfied and eagerly completed after executing *Tr*; the register writes of the success bit of any write exclusives are done eagerly since all write exclusives have promised their success or failure; all non-memory data dependent instructions that feed into the register reads of W have been done eagerly.

   3.2. pop_commit_store_cand and pop_write_co_check must hold.

      3.2.1. All po-earlier dmb `sy`, `isb`, dmb `ld`, dmb `st` are finished.

      3.2.2. If W is a release then all po-earlier reads and writes are finished.

      3.2.3. All po-earlier read acquires are finished.

      3.2.4. commitDataflow holds.

      3.2.5. commitControlflow holds.

      3.2.6. pop_write_co_check holds.

      3.2.7. If W is a successful store exclusive *WE* paired with a load exclusive *RE*, then *RE* is finished, and if *RE* read from a same-thread write W', that write is propagated.

      3.2.1. It is [F];po;[W] $\subseteq$ BC_WC $\subseteq$ Order $\subseteq S$. So by construction of *Tr* all po-earlier dmb `sy`, `isb`, dmb `ld`, dmb `st` are committed, and therefore eagerly finished.

3.2.2. Assume W is a write release and let W' be a po-earlier write. Then by $(W',W) \in \mathsf{[W];po;[Rel]} \subseteq \mathsf{WC\_WC} \subseteq S$ and by construction of *Tr*, the write W' is propagated and therefore eagerly finished. Let R be a program-order earlier read. Then by $(R,W) \in \mathsf{[R];po;[Rel]} \subseteq \mathsf{RC\_WC} \subseteq S$, and by construction of *Tr* the read R is satisfied. Moreover, by construction of *Tr* all reads R' with $(R',R) \in \mathsf{RC\_RC+}$ are satisfied, since it is $(R',R);(R,W) \in \mathsf{RC\_RC+;RC\_WC} \subseteq S$, and writes E' are propagated and barriers E' are committed and E' thus eagerly finished for all $(E',R) \in \mathsf{(WC\_RC|BC\_RC);RC\_RC^{*}}$, since $(E',R);(R,W) \in \mathsf{(WC\_RC|BC\_RC);RC\_RC^{*};RC\_WC} \subseteq S$. Then by Lemma 8 R is finished.

3.2.3. Let R be a read acquire po-before W. Then $(R,W) \in \mathsf{[Acq];po;[W]} \subseteq \mathsf{RC\_WC} \subseteq S$. So by construction of *Tr* the read R is satisfied, all reads R' with $(R',R) \in \mathsf{RC\_RC+}$ are satisfied, by $(R',R);(R,W) \in \mathsf{RC\_RC+;RC\_WC} \subseteq S$, and all writes E' are propagated and barriers E' are committed and E' therefore eagerly finished for $(E',R) \in \mathsf{(WC\_RC|BC\_RC);RC\_RC^{*}}$, since $(E',R);(R,W) \in \mathsf{(WC\_RC|BC\_RC);RC\_RC^{*};RC\_WC}$. Then by Lemma 8, R is finished.

3.2.4. The memory reads feeding into the register reads of W have to be finished. Let R be one such read. Then the read value of R feeds into the address or data of W and have $(R,W) \in \mathsf{[R];(addr|data);[W]} \subseteq \mathsf{RC\_WC} \subseteq S$. Then by construction R is satisfied and eagerly completed, all reads $(R',R) \in \mathsf{RC\_RC+}$ are satisfied since $(R',R);(R,W) \in \mathsf{RC\_RC+;RC\_WC}$ and writes E are propagated and barriers E committed and E therefore eagerly finished for $(E,R) \in \mathsf{(WC\_RC \mid BC\_RC);RC\_RC^{*}}$ since $(E,R);(R,W) \in \mathsf{(WC\_RC \mid BC\_RC);RC\_RC^{*};RC\_WC}$. Then by Lemma 8 R is finished.

3.2.5. Conditional branches po-before W have to be finished.

Assume there is an unfinished branch instruction po-before W, let *BR* be the po-earliest one. The finish transition for *BR* is taken eagerly, so if *BR* is unfinished, then it is because *BR* cannot finish yet.

Since *BR* is the po-earliest unfinished branch its control flow is finished. So it must be the dataflow of *BR* that is unfinished: there is at least one read R that feeds into the register reads of *BR* register reads that is unfinished. But then $(R,W) \in \mathsf{[R];ctrl;[W]} \subseteq \mathsf{RC\_WC} \subseteq S$. So by construction of *Tr* the read R is satisfied, all writes E are propagated and barriers E are committed for $(E,R) \in \mathsf{(WC\_RC \mid BC\_RC);RC\_RC^{*}}$ are therefore eagerly finished, since $(E,R);(R,W) \in \mathsf{(WC\_RC \mid BC\_RC);RC\_RC^{*};RC\_WC} \subseteq S$, and all reads R' with $(R',R) \in$ RC_RC+ are satisfied, since $(R',R);(R,W) \in$ RC_RC*; RC_WC $\subseteq S$. So by Lemma 8 R is finished.

3.2.6. pop_write_co_check holds.

3.2.6.1. All program-order previous same-address writes have to be propagated. Have [W];po-loc;[W] $\subseteq$ WC_WC $\subseteq S$, so all po-earlier writes to the address of W are propagated and therefore eagerly finished.

3.2.6.2. All po-previous memory accesses have their address-feeding memory reads finished and are initiated. Let R be such an address-feeding read. Then $(R,W) \in$ [R];addr;po;[W] $\subseteq$ RC_WC $\subseteq S$. So by construction R is satisfied, all reads R' with $(R',R) \in$ RC_RC+ are satisfied, since $(R',R);(R,W) \in$ RC_RC+; RC_WC $\subseteq S$, and all writes E' are propagated and barriers E' are committed and thus E' eagerly finished for all E' with $(E',R) \in$ (WC_RC|BC_RC); RC_RC*, since for these it is $(E',R);(R,W) \in$ (WC_RC | BC_RC); RC_RC*; RC_WC. Then by Lemma 8 R is finished. Moreover, since all such R are satisfied, all memory accesses po-before W have eagerly done the register reads necessary to determine their address and have eagerly initiated.

3.2.6.3. All po-previous memory reads to the same address must be satisfied, and not restartable. Let R be one such read. Then it is $(R,W) \in$ fr $\subseteq S$ and by construction of S the read R is satisfied. (By 3.2.6.1. have that all same-address writes po-before W are propagated.) Remains to show that all po-earlier memory reads to the same address are not-restartable. Show instead that they are all already finished by induction on the program order prefix of W.

Induction start. For the empty program-order prefix all reads to the location of W are trivially finished.

Induction hypothesis. Assume the statement holds for *Prefix*, show appending an instruction *II* to the prefix preserves the statement. Only have to show that if *II* is a same-address read it is finished. For R to finish the following have to hold:

- commitDataflow. The memory reads feeding into the register writes R reads from have to be finished. Since the register reads of R are all used to determine the address of R these memory reads are all finished by proof of (3.2.6.2.).

- commitControlflow. Since any instruction that R is control-flow dependent on, W is also control-flow dependent on, these instructions have to be finished by proof of (3.2.5.).
- All po-earlier `dmb sy`, `dmb ld`, `isb` are finished. By proof of 3.2.1. all po-earlier `dmb sy`, `dmb ld`, `isb` are finished.
- All po-earlier acquire reads are finished. By proof of (3.2.3.) all po-earlier read acquires are finished.
- If R is an acquire then all po-earlier releases are finished. If R is an acquire then it has to be finished by proof of (3.2.3.).
- If the closest po-earlier write W' to the same address was forwarded to R the memory reads feeding into the register reads of W' must be finished. As proved above all po-earlier writes W' to the same address as W are propagated, hence committed, which includes having the memory reads feeding into W' finished.
- If the closest po-earlier write W' to the same address was not forwarded to R it must be propagated. As proved above all po-earlier writes W' to the same address as W are propagated.
- All memory accesses between W' and R have their address-feeding memory reads finished. As proved for 3.2.6.2. all memory accesses po-before W have their address-feeding memory reads finished.
- All reads R' to the same address between W' and R must be non-restartable. Since all these reads are po-before W and to the same address as W by the induction hypothesis they are finished, so not restartable anymore.

Now since R is satisfied it can be completed. Since pop_finish_load_cand holds R can finish. Since memory-read-finish transitions are eager, R is finished.

3.2.6.4. Any read R that was partially satisfied from W must have requested its unsatisfied slices from storage. By assumption all memory accesses have the same size, so if R was partially satisfied from W it is completely satisfied and there are no such unsatisfied slices.

3.2.7. Assume W is a successful store exclusive WE that is paired with a load exclusive RE. Then RE must be po-before WE. By proof of 3.2.6.3. all po-earlier memory reads to the same address are already finished, so RE finished. Assume RE read from a same-thread write W'. Then it is $(W', RE) \in \text{po}$ and $(RE, W) \in \text{po}$. And since W' and W have the same address, by proof of 3.2.6.1. the write W' is propagated.

3.3. Assume there exists (RE',[(W',_)]) ∈ flat_ss_exclusive_reads after Tr' where RE' and W are to the same address but from different threads.

Then by 0. of the induction hypothesis it is $(W', RE') \in$ rf and there exists a write exclusive WE' such that $(RE', WE') \in$ rmw, $W' \in S[0..n]$, and WE' not in $S[0..n]$.

Since RE' and W from different threads by assumption, also W and WE' from different threads, so $WE' \neq W$ and WE' not in $S[0..n+1]$ either, and therefore $(W, WE') \in S$. Since it is $W' \in S[0..n]$ it is also $(W', W) \in S$. So $(W', W) \in S$ and $(W, WE') \in S$ where W and WE' are from different threads but to the same address. Contradiction to Lemma 5 for $(RE', WE') \in$ rmw.

Case E is a barrier B. Show extending the trace for E preserves properties 0. – 3..

0. Committing a barrier does not change flat_ss_exclusive_reads, and since E is not a write, 0. still holds.
1. By induction hypothesis the trace Tr for E induces the po, rf, co, and rmw relations from the candidate execution. Since B does not fetch, satisfy reads, propagate writes, or promise the success/failure of write exclusives this is still true for Tr'.
2. Have to show that committing B does not restart any instructions or discard instruction-tree branches. But this follows from pop_commit_barrier_action's definition.
3. Tr' is a valid trace of Flat Operational. Have to show that committing B is enabled after Tr.

3.1. commitDataflow. Since B has no data this is vacuously true.

3.2. commitControlflow.

Conditional branches po-before B have to be finished. Assume there is an unfinished branch instruction po-before B, let BR be the *po*-earliest one. The finish transition for BR is taken eagerly, so if BR is unfinished, then it is because BR cannot finish yet.

Since BR is the *po*-earliest unfinished branch its control flow is finished. So it must be BR's dataflow that is unfinished: there is at least one read R that feeds into the register reads of BR register reads that is unfinished. But then $(R, B) \in$ [R];ctrl;[F] ⊆ RC_BC ⊆ S. So by construction of Tr the read R is satisfied, all writes E are propagated and barriers E are committed and E therefore eagerly finished for $(E, R) \in$ (WC_RC | BC_RC); RC_RC*, since $(E, R); (R, B) \in$ (WC_RC | BC_RC); RC_RC*; RC_BC, and all reads R' with $(R', R) \in$ RC_RC+ are satisfied, since $(R', R); (R, B) \in$ RC_RC*; RC_BC ⊆ S. Then R is finished.

3.3. If B is a `dmb sy` all barriers, reads, and writes, are finished.

Let B' be a barrier po-before B. Then $(B',B) \in [\mathsf{F}];\mathsf{po};[\mathsf{DMB.SY}] \subseteq \mathsf{BC\_BC} \subseteq S$, so by construction of *Tr* the barrier B is committed and therefore eagerly finished. Let W be a po-earlier write. then $(W,B) \in [\mathsf{W}];\ \mathsf{po};\ [\mathsf{DMB.SY}] \subseteq \mathsf{WC\_BC} \subseteq S$. So by construction of *Tr* the write W is propagated and therefore eagerly completed and finished.

Let R be a po-earlier read. So $(R,B) \in [\mathsf{R}];\mathsf{po};[\mathsf{DMB.SY}] \subseteq \mathsf{RC\_BC} \subseteq S$. So by construction R is satisfied, all writes E are propagated and barriers E are committed and E thus eagerly finished for $(E,R) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC});\ \mathsf{RC\_RC}^*$, since $(E,R);(R,B) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC});\ \mathsf{RC\_RC}^*;\ \mathsf{RC\_BC} \subseteq S$, and all reads R' with $(R',R) \in \mathsf{RC\_RC+}$ are satisfied, since $(R',R);(R,B) \in \mathsf{RC\_RC+};\ \mathsf{RC\_BC}$. Then by Lemma 8 R is finished.

3.4. All po-earlier `dmb sy` are finished. Let B' be a `dmb sy` po-before B. Then it is $(B',B) \in [\mathsf{DMB.SY}];\mathsf{po};[\mathsf{F}] \subseteq \mathsf{BC\_BC} \subseteq S$. So by construction of *Tr*$'$ the barrier B' is committed and therefore eagerly finished.

3.5. If B is an `isb` all po-earlier memory accesses have their address-feeding memory reads finished and have initiated.

Let R be a memory read feeding into the address of a po-earlier memory access. Then $(R,B) \in [\mathsf{R}];\ \mathsf{addr};\ \mathsf{po};\ [\mathsf{ISB}] \subseteq \mathsf{RC\_BC} \subseteq S$. So by construction R is satisfied, all writes E are propagated and barriers E committed and E therefore eagerly finished for $(E,R) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC});\ \mathsf{RC\_RC}^*$, since $(E,R);(R,B) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC});\ \mathsf{RC\_RC}^*;\ \mathsf{RC\_BC} \subseteq S$, and all reads R' with $(R',R) \in \mathsf{RC\_RC+}$ are satisfied, since $(R',R);(R,B) \in \mathsf{RC\_RC+};\ \mathsf{RC\_BC}$. Then by Lemma 8 R is finished. Moreover, since all such R are satisfied, all memory accesses po-before B have eagerly done the register reads necessary to determine their address and have eagerly initiated.

3.6. If B is a `dmb ld` all po-earlier memory loads are finished. Let R be a po-earlier read. Then $(R,B) \in [\mathsf{R}];\mathsf{po};[\mathsf{DMB.LD}] \subseteq \mathsf{RC\_BC} \subseteq S$. So by construction R is satisfied, all writes E are propagated and barriers E are committed and E therefore eagerly finished for $(E,R) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC});\ \mathsf{RC\_RC}^*$, since $(E,R);(R,B) \in (\mathsf{WC\_RC} \mid \mathsf{BC\_RC});\ \mathsf{RC\_RC}^*;\ \mathsf{RC\_BC} \subseteq S$, and all reads R' with $(R',R) \in \mathsf{RC\_RC+}$ are satisfied, since $(R',R);(R,B) \in \mathsf{RC\_RC+};\ \mathsf{RC\_BC}$. Then by Lemma 8 R is finished.

3.7. If B is a `dmb st` all po-earlier memory stores are finished.

Let W be a po-earlier write. Then $(W,B) \in [\mathsf{W}];\mathsf{po};[\mathsf{DMB.ST}] \subseteq \mathsf{WC\_BC} \subseteq S$. So by construction of *Tr* the write W is propagated and therefore eagerly finished.

Take eager transitions. Repeatedly extend the trace Tr' to Tr'' for enabled eager transitions T, until there are no more enabled eager transitions.

0. Only the promise-write-success, satisfy-read-in-memory and propagate-memory-write transitions change the flat_ss_exclusive_reads field. These are not eager, so flat_ss_exclusive_reads unchanged. And since the prefix of S has not changed: 0. still holds.
1. By definition of transition-eagerness, T does not fetch, satisfy a read, propagate a write, or promise success or failure of a write exclusive, so 1. is preserved.
2. Restarts are caused only by the transitions promise-write-exclusive-failure, satisfy-read-by-forwarding, satisfy-read-from-memory, and propagate-memory-write. By definition these transitions are not eager, so T does not cause restarts. Only finishing of branch instructions causes instruction tree branches to be discarded. A branch BR can only finish when the memory reads feeding into its register reads are finished. Let R be any such memory read. If $R = S[n+1]$ (R was the last event from S to be handled before the eager steps) then R reads from the unique W such that $(W, R) \in \mathsf{rf}$ by proof of 1.; for all other R the induced reads-from relation is a subset of rf of the candidate execution by induction hypothesis. So the successor of BR is determined as in the candidate execution's po, and by induction hypothesis the instruction tree viewed as a relation matches po. Therefore, finishing such a branch BR does not discard any instruction tree branches.
3. Since by assumption T is enabled after Tr', Tr'' is a valid trace.
4. When no more eager transitions are enabled, establish property 4.: the eager transitions have all been taken by construction.

□

4 ARMv8 Axiomatic behaviour included in Flat Axiomatic

Theorem 4. *Let C be a candidate execution accepted by ARMv8-axiomatic. Then C is accepted by Flat-axiomatic*

Proof. Since the Internal and Atomic axioms are the same in both models, only have to show that when ARMv8-axiomatic's axioms hold, then Flat-axiomatic's Order acyclic. To do this, start with Order, and show step-by-step that any edges in Order that are not included in ARMv8-axiomatic's ob are can be deleted safely: if there is a cycle in Order with them, then there is also one in the relation without them.

(Order)+
= (BC_RS
| WC_RS
| RS_RS
| RS_RC; RC_RC*; (RC_BC | RC_WC)
| WC_RC; RC_RC*; (RC_BC | RC_WC)
| BC_RC; RC_RC*; (RC_BC | RC_WC)
| BC_BC
| RC_BC
| WC_BC
| RC_WC
| WC_WC
| BC_WC
| co
| rfe
| fr
)+

Apply most of the definitions.

...
= ([DMB.SY|**ISB**|DMB.LD]; po; [**R**]
| [Rel];po;[Acq]
| [Xw];rfi;[Acq]
| [**W**];(po—loc \ rf \ (po;rf));[**R**]
| [Acq];po;[**R**]
| [**R**];addr;[**R**]
| [**R**];(addr|data);rfi;[**R**]
| [**R**];(po—loc \ (rf^{-1};rf) \ (po;rf));[**R**]
| RC_RC*; (RC_BC | RC_WC)
| WC_RC; RC_RC*; (RC_BC | RC_WC)
| BC_RC; RC_RC*; (RC_BC | RC_WC)
| [DMB.SY];po;[F]
| [F];po;[DMB.SY]
| [**R**];po;[DMB.SY|DMB.LD]
| [**R**];ctrl;[F]
| [**R**];addr;po;[**ISB**]
| [**W**];po;[DMB.SY|DMB.ST]
| [**R**];po;[Rel]
| [**R**];addr;[**W**]
| [**R**];data;[**W**]
| [**R**];ctrl;[**W**]
| [**R**];addr;po;[**W**]
| [Acq];po;[**W**]
| [**R**];po—loc;[**W**]
| [**R**];rmw;[**W**]
| [**W**];po—loc;[**W**]
| [**W**];po;[Rel]
| [F];po;[**W**]
| co
| rfe
| fr
)+

Simplify.

...
= ([DMB.SY];po
| po;[DMB.SY]
| [F];po;[**W**]
| [**R**];po;[DMB.LD]

```
    | [W];po;[DMB.ST]
    | [R];addr;po;[ISB]
    | [ISB|DMB.LD]; po; [R]
    | ctrl;[F]
    | po;[Rel]
    | [Acq];po;[R|W]
    | [Rel];po;[Acq]
    | [Xw];rfi;[Acq]
    | rmw
    | addr
    | data
    | ctrl;[W]
    | addr;po;[W]
    | (addr|data);rfi
    | RC_RC*; (RC_BC | RC_WC)
    | WC_RC; RC_RC*; (RC_BC | RC_WC)
    | BC_RC; RC_RC*; (RC_BC | RC_WC)
    | co
    | rfe
    | fr
    | [R];(po—loc \ (rf^{-1};rf) \ (po;rf));[R]
    | [W];(po—loc \ rf \ (po;rf));[R]
    | [W];po—loc;[W]
    | [R];po—loc;[W]
    )+
```

[W];po-loc;[W] is included in co, so can delete this edge. [R];po-loc;[W] included in fr, so can delete this edge.

```
  ...
  = ([DMB.SY];po
    | po;[DMB.SY]
    | [F];po;[W]
    | [R];po;[DMB.LD]
    | [W];po;[DMB.ST]
    | [R];addr;po;[ISB]
    | [ISB|DMB.LD]; po; [R]
    | ctrl;[F]
    | po;[Rel]
    | [Acq];po;[R|W]
    | [Rel];po;[Acq]
    | [Xw];rfi;[Acq]
    | rmw
    | addr
    | data
    | ctrl;[W]
    | addr;po;[W]
    | (addr|data);rfi
    | RC_RC*; (RC_BC | RC_WC)
    | WC_RC; RC_RC*; (RC_BC | RC_WC)
    | BC_RC; RC_RC*; (RC_BC | RC_WC)
    | co
    | rfe
    | fr
    | [R];(po—loc \ (rf^{-1};rf) \ (po;rf));[R]
    | [W];(po—loc \ rf \ (po;rf));[R]
    )+
```

Consider $(W,R) \in$ [W];(po-loc\rf\(po;rf));[R]. By definition $(W,R) \notin$ rf. Let $(W',R) \in$ rf with $W \neq W'$. By definition it is not $(W,W')(W',R) \subseteq$ po. By coherence axiom it also cannot be

$(W', W) \in$ po, because otherwise cycle in fr;po. Also cannot be $(R, W') \in$ po because otherwise cycle in po;rf. So $(W', R) \in$ rfe, and by coherence axiom again it must be $(W, W') \in$ co, otherwise cycle in fr;po. So then $(W, R) \in$ co;rfe. So the above edge is subsumed by co;rfe.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | RC_RC*; (RC_BC | RC_WC)
  | WC_RC; RC_RC*; (RC_BC | RC_WC)
  | BC_RC; RC_RC*; (RC_BC | RC_WC)
  | co
  | rfe
  | fr
  | [R];(po—loc \ (rf^{-1};rf) \ (po;rf));[R]
  )+
```

Consider $(R, R') \in$ [R];(po-loc\ (rf^{-1};rf)\(po;rf));[R] and let $(W, R) \in$ rf, $(W', R') \in$ rf. By definition $W \neq W'$ and not $(R, W'), (W', R') \subseteq$ po. It must be $(W, W') \in$ co as otherwise $(R', W) \in$ fr and there is a cycle in fr;rf;po.

By per-thread-coherence it cannot be $(R', W') \in$ po. Also cannot be $(W', R) \in$ po, since otherwise cycle in fr;po. So W' not from the same thread as R and R' and it is $(R, R') \in$ fr;rfe. So [R];(po-loc \ (rf^{-1};rf) \ (po;rf));[R] subsumed by fr;rfe and can delete the edge.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
```

```
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | RC_RC*; (RC_BC | RC_WC)
  | WC_RC; RC_RC*; (RC_BC | RC_WC)
  | BC_RC; RC_RC*; (RC_BC | RC_WC)
  | co
  | rfe
  | fr
  )+
```

Now apply the definitions of WC_RC and BC_RC.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | RC_RC*; (RC_BC | RC_WC) (* E2 *)
  | ([Rel];po;[Acq] | [W];(po—R—loc\rf);[R]); RC_RC*; (RC_BC | RC_WC)
  | [DMB.SY|ISB|DMB.LD];po;[R]; RC_RC*; (RC_BC | RC_WC) (* E1 *)
  | co
  | rfe
  | fr
  )+
```

Have E1 = [DMB.SY|ISB|DMB.LD];po;[R];E2 [DMB.SY|ISB|DMB.LD];po;[R] already contained in the relation using [DMB.SY];po and [ISB|DMB.LD];po;[R]. So E1 is already contained in the relation using [DMB.SY];po, [ISB|DMB.LD];po;[R], and E2, and can delete E1.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
```

```
| addr
| data
| ctrl;[W]
| addr;po;[W]
| (addr|data);rfi
| RC_RC*; (RC_BC | RC_WC) (* E2 *)
| ([Rel];po;[Acq] | [W];(po—R—loc\rf);[R]); RC_RC*; (RC_BC | RC_WC) (* E1 *)
| co
| rfe
| fr
)+
```

Have E1 = ([Rel];po;[Acq] | [W];(po-R-loc\rf);[R]);E2. Now assume W and R such that $(W,R) \in$ [W];(po-R-loc \ rf);[R]. By definition R does not read from W, so $(W',R) \in$ rf for $W' \neq W$. W and W' have to be coherence related. It cannot be $(W',W) \in$ co, because then there is a cycle in po;fr. So have $(W,W') \in$ co.

By $(W,W') \in$ co it cannot be $(W',W) \in$ po. By $(W',R) \in$ rf it cannot be $(R,W') \in$ po. And by definition of po-R-loc it cannot be $(W,W'),(W',R) \in$ po. So W' not from the same thread as R and it is $(W',R) \in$ rfe. Therefore it is $(W,W');(W',R) \in$ co;rfe, so $(W,R) \in$ co;rfe. So [W];(po-R-loc\rf);[R] is included in co;rfe.

Then ([Rel];po;[Acq] | [W];(po-R-loc\rf);[R]) is already included in the above relation using [Rel];po;[Acq] and co and rfe, and E1 is subsumed by the combination of these and E2. So can delete E1.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | RC_RC*; (RC_BC | RC_WC)
  | co
  | rfe
  | fr
  )+
```

Now consider RC_RC*:

```
([R];addr;[R] | [R];addr;po—no—W—loc;[R] | [Acq];po;[R] | [R];ctrl;[R] |
[R];(addr|data);rfi;[R] | [R];po—R—loc;[R])*
```

```
= (addr;[R] | addr;po—no—W—loc;[R] | [Acq];po;[R] | ctrl;[R] |
   (addr|data);rfi;[R] | [R];po—R—loc;[R])*
```

This is included in the following:

```
(addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
   (addr|data);rfi;[R] | [R];po—R—loc;[R])*.
```

Can rewrite this to the following, using the fact that [R];po-R-loc;[R] is transitive:

```
([R];po—R—loc;[R])?; (addr;[R]
                       | addr;[R];po—R—loc;[R]
                       | addr;po;[R]
                       | addr;po;[R];po—R—loc;[R]
                       | [Acq];po;[R]
                       | [Acq];po;[R];po—R—loc;[R]
                       | ctrl;[R]
                       | ctrl;[R];po—R—loc;[R]
                       | (addr|data);rfi;[R]
                       | (addr|data);rfi;[R];po—R—loc;[R])*
```

But some edges are subsumed by others:

- addr;[R];po-R-loc;[R] by addr;po;[R],
- addr;po;[R];po-R-loc;[R] by addr;po;[R],
- [Acq];po;[R];po-R-loc;[R] by [Acq];po;[R],
- ctrl;[R];po-R-loc;[R] by ctrl;[R].

So rewrite to:

```
([R];po—R—loc;[R])?; (addr;[R]
                        | addr;po;[R]
                        | [Acq];po;[R]
                        | ctrl;[R]
                        | (addr|data);rfi;[R]
                        | (addr|data);rfi;[R];po—R—loc;[R])* =
```

So RC_RC* included in:

```
([R];po—R—loc;[R])?; (addr;[R] | addr;po;[R] | [Acq];po;[R] |
   ctrl;[R] | (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*
```

So can strengthen the order below by including this edge instead.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
```

```
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
        (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*; (RC_BC | RC_WC)
  | co
  | rfe
  | fr
  )+
```

Apply the definition of RC_BC and RC_WC.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
         (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
            ([R];po;[DMB.SY|DMB.LD] | [R];ctrl;[F] | [R];addr;po;[ISB] |
             [R];po;[Rel] | [R];addr;[W] | [R];data;[W] | [R];ctrl;[W] |
             [R];addr;po;[W] | [Acq];po;[W] | [R];po—loc;[W] | [R];rmw;[W]) (* E *)
  | co
  | rfe
  | fr
  )+
```

Some cases of E are subsumed by other edges in the relation, so can delete these: set of edges ending in [R];po;[DMB.SY|DMB.LD] is subsumed by po;[DMB.SY] and [R];po;[DMB.LD]; the set of edges ending with [R];po;[Rel] is subsumed by po;[Rel].

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
```

```
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
      (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
        ([R];ctrl;[DMB.ST|ISB] | [R];addr;po;[ISB] | [R];addr;[W] |
         [R];data;[W] | [R];ctrl;[W] | [R];addr;po;[W] | [Acq];po;[W] |
         [R];po—loc;[W] | [R];rmw;[W]) (* E *)
  | co
  | rfe
  | fr
  )+
```

The set of edges E is a subset of program order and cannot create cycles by itself. So it can only create cycles in composition with other edges. In particular, the subset of E ending with [DMB.ST] can only create cycles in composition with others. Since E cannot be composed with itself, replace it with its post-composition with every other edge from the relation above.

```
...
only has a cycle if the following has a cycle
  ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
      (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
        ([R];ctrl;[DMB.ST];(po;[DMB.SY]|po;[W]|po;[Rel]) | [R];ctrl;[ISB] |
         [R];addr;po;[ISB] | [R];addr;[W] | [R];data;[W] | [R];ctrl;[W] |
         [R];addr;po;[W] | [Acq];po;[W] | [R];po—loc;[W] | [R];rmw;[W]) (* E *)
  | co
  | rfe
  | fr
  )+
```

The definition of E contains some duplication, so simplify.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD]; po; [R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
        (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
          ([R];ctrl;[DMB.ST];po;[DMB.SY] | [R];ctrl;[ISB] | [R];addr;po;[ISB] |
           [R];addr;[W] | [R];data;[W] | [R];ctrl;[W] | [R];addr;po;[W] |
           [Acq];po;[W] | [R];po—loc;[W] | [R];rmw;[W]) (* E *)
  | co
  | rfe
  | fr)+
```

Now the subset of E ending with DMB.SY is subsumed by the edges po;[DMB.SY], so can delete it. Also rewrite [R];po-loc;[W] to fri (by Internal axiom).

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | ([R];po—R—loc;[R])?; [R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
        (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*; ([R];ctrl;[ISB] |
         [R];addr;po;[ISB] | [R];addr;[W] | [R];data;[W] | [R];ctrl;[W] |
         [R];addr;po;[W] | [Acq];po;[W] | fri | [R];rmw;[W]) (* E *)
  | co
  | rfe
  | fr)+
```

Split E by definition of the ‘?’ operator.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | [R];(addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi;[R] |
       (addr|data);rfi;[R];po—R—loc;[R])*;
         ([R];ctrl;[ISB] | [R];addr;po;[ISB] | [R];addr;[W] | [R];data;[W] |
         [R];ctrl;[W] | [R];addr;po;[W] | [Acq];po;[W] | fri |
         [R];rmw;[W]) (* E1 *)
  | [R];po—R—loc;[R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
       (addr|data);rfi;[R] | (addr|data);rfi;[R];po—R—loc;[R])*;
         ([R];ctrl;[ISB] | [R];addr;po;[ISB] | [R];addr;[W] | [R];data;[W] |
          [R];ctrl;[W] | [R];addr;po;[W] | [Acq];po;[W] | fri |
          [R];rmw;[W]) (* E2 *)
  | co
  | rfe
  | fr
  )+
```

Now consider $(W,R') \in$ rfi;[R];po-R-loc;[R]. Then there exists a read R such that $(W,R) \in$ rfi and $(R,R') \in$ po-R-loc. Let $(W',R') \in$ rf. Now there are two cases $W = W'$ or $W \neq W'$.

$W = W'$ Then $(W,R') \in$ rfi.

$W \neq W'$ W and W' must be coherence-related. Assume $(W',W) \in co$. Then $(R',W) \in$ fr and there is a cycle in po;fr. So $(W,W') \in$ co. It cannot be $(W',R) \in$ po because otherwise cycle in po;fr, and it cannot be $(R',W') \in$ po because otherwise cycle in po;rf. By definition of po-R-loc, W' not po-between R and R'. So W' and R' from different threads and it is $(W',R') \in$ rfe. Therefore $(W,W');(W',R') \in$ co;rfe, so $(W,R') \in$ co;rfe.

So rfi;[R];po-R-loc;[R] included in rfi | (co;rfe) & po-loc. Use this to strengthen E1 and E2.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
```

```
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | [R];(addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1 *)
  | [R];po—R—loc;[R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
       (addr|data);rfi | (addr|data);((co;rfe) & po—loc))*; [R]; (ctrl;[ISB] |
       addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] | addr;po;[W] |
       [Acq];po;[W] | fri | rmw;[W])
  | co
  | rfe
  | fr
  )+
```

Simplify: E1 is subsumed by the combination of the following edge sets:

- addr;po;[ISB],
- addr;po;[W],
- [Acq];po;[R],
- ctrl;[ISB],
- [Acq];po;[W],
- ctrl;[W],
- addr;[W],
- data;[W],
- fr,
- rmw,
- (addr|data);rfi,
- co,
- rfe,
- rmw.

So can drop E1.

```
…
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
```

```
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | [R];po—R—loc;[R]; (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] |
      (addr|data);rfi | (addr|data);((co;rfe) & po—loc))*; [R];
        (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
         addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
  | co
  | rfe
  | fr
  )+
```

The edge set E2 cannot create cycles by itself since it is a subset of program order (which in turn is acyclic). Any cycle contained in the order above that has a cycle using an edge from E2 must be one that uses it in composition with more edges from the relation. E2 does not compose with itself. So it suffices to replace E2 by the post-composition of all other edges with this one.

```
...
only has a cycle if the following has a cycle
  ([DMB.SY];po
  | [DMB.SY];po;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1*)
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | [ISB|DMB.LD];po;[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Acq];po;[R|W];[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
        addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E3 *)
  | [Rel];po;[Acq]
  | [Rel];po;[Acq];[R];po—R—loc;[R];
      (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
        (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
         addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E4 *)
  | [Xw];rfi;[Acq]
```

```
| [Xw];rfi;[Acq];[R];po—R—loc;[R];
     (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
        (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
         addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E5 *)
| rmw
| addr
| addr;[R];po—R—loc;[R];
     (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
        (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
         addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E6 *)
| data
| ctrl;[W]
| addr;po;[W]
| (addr|data);rfi
| (addr|data);rfi;[R];po—R—loc;[R];
     (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
        (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
         addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
| co
| rfe
| rfe;[R];po—R—loc;[R];
     (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
        (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
         addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
| fr)+
```

Some of these are easily subsumed by existing edges:

- E1 by [DMB.SY];po
- E2 by [F];po;[W] and [ISB|DMB.LD];po;[R] and ctrl;[ISB] | addr;po;[ISB]
- E3 by [Acq];po;[W] and [Acq];po;[R] and ctrl;[ISB] | addr;po;[ISB]
- E4 by the sets [Rel];po;[Acq] and [Acq];po;[W] and [Acq];po;[R] and ctrl;[ISB] and addr;po;[ISB].
- E5 by the sets [Xw];rfi;[Acq] and [Acq];po;[W] and [Acq];po;[R] and ctrl;[ISB] and addr;po;[ISB].
- E6 is subsumed by addr;po;[ISB] and addr;po;[W].

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
```

```
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | (addr|data);rfi;[R];po—R—loc;[R];
        (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
          (addr|data);((co;rfe) & po—loc))*; [R];
            (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
              addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1 *)
  | co
  | rfe
  | rfe;[R];po—R—loc;[R];
        (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
          (addr|data);((co;rfe) & po—loc))*; [R];
            (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
              addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
  | fr
  )+
```

As shown before, rfi;[R];po-R-loc;[R] included in (rfi | ((co;rfe)&po-loc)). Use this to strengthen E1.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | (addr|data);(rfi | ((co;rfe) & po—loc));[R];
        (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
          (addr|data);((co;rfe) & po—loc))*; [R];
            (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
              addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
  | co
  | rfe
  | rfe;[R];po—R—loc;[R];
        (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
            (addr|data);((co;rfe) & po—loc))*; [R];
              (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
                addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
  | fr)+
```

Now consider $(W,R') \in$ rfe;[R];po-R-loc;[R]. Then $(W,R) \in$ rfe for some read R and $(R,R') \in$ po-R-loc. Now there are two cases: $(W,R') \in$ rf or otherwise.

$(W,R') \in$ rf Then (W,R') in rfe, by Internal axiom.

otherwise Then there is a write W' with $(W',R') \in$ rf and $W \neq W'$. W and W' must be coherence-related. Assume the coherence is $(W',W) \in$ co. Then it is $(R',W) \in$ fr and there is a cycle fr;rf;po. So the coherence must be $(W,W') \in$ co. It cannot be $(W',R) \in$ po because then there would be a cycle in po;fr. By definition of the edge, W' not po-between R and R'. And it cannot be $(R',W') \in$ po since then there would be a cycle in po;rf. So W' not from the same thread as R and R'. But then it is $(W,R);(R,W');(W',R') \in$ rfe;fre;rfe.

So rfe;[R];po-R-loc;[R] included in (rfe | rfe;fre;rfe). Use this to strengthen E2.

```
…
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | (addr|data);(rfi | ((co;rfe) & po—loc));[R];
        (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
         (addr|data);((co;rfe) & po—loc))*; [R];
           (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
            addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
  | co
  | rfe
  | (rfe | rfe;fre;rfe);
        (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
         (addr|data);((co;rfe) & po—loc))*; [R];
           (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
            addr;po;[W] | [Acq];po;[W] | fri | rmw;[W])
  | fr
  )+
```

Split the first and the second long edge.

```
…
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
```

```
| [Acq];po;[R|W]
| [Rel];po;[Acq]
| [Xw];rfi;[Acq]
| rmw
| addr
| data
| ctrl;[W]
| addr;po;[W]
| (addr|data);rfi
| (addr|data);rfi;[R];
     (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1 *)
| (addr|data);((co;rfe) & po—loc);[R];
     (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
| co
| rfe
| rfe;
     (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E3 *)
| rfe;fre;rfe;
     (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
      (addr|data);((co;rfe) & po—loc))*; [R];
         (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
          addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E4 *)
| fr
)+
```

Now check E1 – E4, each with a different prefix.

- E1 starting wt (addr|data);rfi. Since already have (addr|data);rfi in the relation can strengthen the order by dropping this prefix.
- E2 starts with (addr|data);((co;rfe)&po-loc). Since have the edges addr, data, co, rfe, can delete this prefix and strengthen the order.
- E3 starts with rfe. Since have rfe in the relation, can strengthen the order by deleting this prefix.
- E4 starts with rfe;fre;rfe. Since rfe and fr are already in the relation, can strengthen it by deleting this prefix.

```
...
⊆ ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
```

```
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E1 *)
  | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E2 *)
  | co
  | rfe
  | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E3 *)
  | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E4 *)
  | fr
  )+
```

Now have four copies of the same edge, can delete all but one.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | ctrl;[W]
  | addr;po;[W]
  | (addr|data);rfi
  | (addr;[R] | addr;po;[R] | [Acq];po;[R] | ctrl;[R] | (addr|data);rfi |
       (addr|data);((co;rfe) & po—loc))*; [R];
          (ctrl;[ISB] | addr;po;[ISB] | addr;[W] | data;[W] | ctrl;[W] |
           addr;po;[W] | [Acq];po;[W] | fri | rmw;[W]) (* E *)
  | co
  | rfe
  | fr
  )+
```

`E` is subsumed by the combination of other edges already in the relation: addr;po;[ISB] and

[Acq];po;[R] and ctrl;[ISB] and (addr|data);rfi and addr and data and co and rfe and addr;po;[W] and [Acq];po;[W] and ctrl;[W] and addr;[W] and data;[W] and fri and rmw.

```
...
= ([DMB.SY];po
   | po;[DMB.SY]
   | [F];po;[W]
   | [R];po;[DMB.LD]
   | [W];po;[DMB.ST]
   | [R];addr;po;[ISB]
   | [ISB|DMB.LD];po;[R]
   | ctrl;[F]
   | po;[Rel]
   | [Acq];po;[R|W]
   | [Rel];po;[Acq]
   | [Xw];rfi;[Acq]
   | rmw
   | addr
   | data
   | ctrl;[W]
   | addr;po;[W]
   | (addr|data);rfi
   | co
   | rfe
   | fr
   )+
```

Split co and fr.

```
...
= ([DMB.SY];po
   | po;[DMB.SY]
   | [F];po;[W]
   | [R];po;[DMB.LD]
   | [W];po;[DMB.ST]
   | [R];addr;po;[ISB]
   | [ISB|DMB.LD];po;[R]
   | ctrl;[F]
   | po;[Rel]
   | [Acq];po;[R|W]
   | [Rel];po;[Acq]
   | [Xw];rfi;[Acq]
   | rmw
   | addr
   | data
   | ctrl;[W]
   | addr;po;[W]
   | (addr|data);rfi
   | coe
   | coi
   | rfe
   | fre
   | fri
   )+
```

coi is acyclic itself. It can only contribute to cycles in composition with other edges. Postcompose every edge EDGE in the relation with coi and add EDGE;coi.

```
...
only has a cycle if the following has a cycle
  ([DMB.SY];po
  | [DMB.SY];po;coi
  | po;[DMB.SY]
  | [F];po;[W]
  | [F];po;[W];coi
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Acq];po;[R|W];coi
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | rmw;coi
  | addr
  | addr;coi
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | addr;po;[W];coi
  | (addr|data);rfi
  | coe
  | coe;coi
  | rfe
  | fre
  | fre;coi
  | fri
  | fri;coi
  )+
```

Most of those edges are subsumed by others:

- [DMB.SY];po;coi subsumed by [DMB.SY];po,
- [F];po;[W];coi by [F];po;[W],
- [Acq];po;[W];coi by [Acq];po;[W],
- rmw;coi subsumed by fri,
- addr;coi subsumed by addr;po;[W],
- addr;po;[W];coi subsumed by addr;po;[W],
- coe;coi subsumed by coe,
- fre;coi subsumed by fre,
- fri;coi subsumed by fri.

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
```

```
   | [W];po;[DMB.ST]
   | [R];addr;po;[ISB]
   | [ISB|DMB.LD];po;[R]
   | ctrl;[F]
   | po;[Rel]
   | po;[Rel];coi
   | [Acq];po;[R|W]
   | [Rel];po;[Acq]
   | [Xw];rfi;[Acq]
   | rmw
   | addr
   | data
   | data;coi
   | ctrl;[W]
   | ctrl;[W];coi
   | addr;po;[W]
   | (addr|data);rfi
   | coe
   | rfe
   | fre
   | fri
   )+
```

fri is acyclic itself, so can only create cycles in composition with other edges. Post-compose all edges `EDGE` with fri and add `EDGE;fri`.

```
...
only has a cycle if the following has a cycle
   ([DMB.SY];po
   | [DMB.SY];po;fri
   | po;[DMB.SY]
   | [F];po;[W]
   | [R];po;[DMB.LD]
   | [W];po;[DMB.ST]
   | [R];addr;po;[ISB]
   | [ISB|DMB.LD];po;[R]
   | [ISB|DMB.LD];po;[R];fri
   | ctrl;[F]
   | po;[Rel]
   | po;[Rel];coi
   | [Acq];po;[R|W]
   | [Acq];po;[R|W];fri
   | [Rel];po;[Acq]
   | [Rel];po;[Acq];fri
   | [Xw];rfi;[Acq]
   | [Xw];rfi;[Acq];fri
   | rmw
   | addr
   | addr;fri
   | data
   | data;coi
   | ctrl;[W]
   | ctrl;[W];coi
   | addr;po;[W]
   | (addr|data);rfi
   | (addr|data);rfi;fri
   | coe
   | rfe
   | rfe;fri
   | fre
```

```
 )+
```

But these edges are subsumed by others in the relation.

- [DMB.SY];po;fri by [DMB.SY];po,
- [ISB|DMB.LD];po;[R];fri by [F];po;[W],
- [Acq];po;[R|W];fri by [Acq];po;[R|W],
- [Rel];po;[Acq];fri by [Rel];po;[Acq] and [Acq];po;[R|W],
- [Xw];rfi;[Acq];fri by [Xw];rfi;[Acq] and [Acq];po;[R|W],
- addr;fri by addr;po;[W],
- (addr|data);rfi;fri by addr;po;[W] and data;coi,
- rfe;fri by coe.

```
…
= ([DMB.SY];po
  | po;[DMB.SY]
  | [F];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[F]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

Simplify.

```
…
= ([DMB.SY];po
  | po;[DMB.SY]
  | [DMB.LD|ISB];po;[W]
  | [DMB.ST];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[DMB.ST]
  | ctrl;[ISB]
  | po;[Rel]
  | po;[Rel];coi
```

```
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

The edges with DMB.ST themselves are acyclic. Replace them by all possible compositions using them.

```
...
only has a cycle if the following has a cycle
  ([DMB.SY];po
  | po;[DMB.SY]
  | [DMB.LD|ISB];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST];po;[W]
  | [R];addr;po;[ISB]
  | [ISB|DMB.LD];po;[R]
  | ctrl;[DMB.ST];po;[W]
  | ctrl;[ISB]
  | po;[Rel]
  | po;[Rel];coi
  | [Acq];po;[R|W]
  | [Rel];po;[Acq]
  | [Xw];rfi;[Acq]
  | rmw
  | addr
  | data
  | data;coi
  | ctrl;[W]
  | ctrl;[W];coi
  | addr;po;[W]
  | (addr|data);rfi
  | coe
  | rfe
  | fre
  )+
```

The ctrl;[DMB.ST];po;[W] edge is subsumed by ctrl;[W].

```
...
= ([DMB.SY];po
  | po;[DMB.SY]
  | [DMB.LD|ISB];po;[W]
  | [R];po;[DMB.LD]
  | [W];po;[DMB.ST];po;[W]
  | [R];addr;po;[ISB]
  | [ISB];po;[R]
  | [DMB.LD];po;[R]
```

```
   | ctrl;[ISB]
   | po;[Rel]
   | po;[Rel];coi
   | [Acq];po;[R|W]
   | [Rel];po;[Acq]
   | [Xw];rfi;[Acq]
   | rmw
   | addr
   | data
   | data;coi
   | ctrl;[W]
   | ctrl;[W];coi
   | addr;po;[W]
   | (addr|data);rfi
   | coe
   | rfe
   | fre
   )+
```

Merge two ISB edge sets.

```
...
only has a cycle if the following has one
   ([DMB.SY];po
   | po;[DMB.SY]
   | [DMB.LD|ISB];po;[W]
   | [R];po;[DMB.LD]
   | [W];po;[DMB.ST];po;[W]
   | [R];(ctrl|(addr;po));[ISB]
   | [ISB];po;[R]
   | [DMB.LD];po;[R]
   | po;[Rel]
   | po;[Rel];coi
   | [Acq];po;[R|W]
   | [Rel];po;[Acq]
   | [Xw];rfi;[Acq]
   | rmw
   | addr
   | data
   | data;coi
   | ctrl;[W]
   | ctrl;[W];coi
   | addr;po;[W]
   | (addr|data);rfi
   | coe
   | rfe
   | fre
   )+
```

The ISB edge sets themselves are acyclic. Replace them by all possible compositions with other edges.

```
...
only has a cycle if the following has one
   ([DMB.SY];po
   | po;[DMB.SY]
   | [DMB.LD];po;[W|R]
   | [R];po;[DMB.LD]
   | [W];po;[DMB.ST];po;[W]
```

```
   | [R];(ctrl|(addr;po));[ISB];po;[W|R]
   | po;[Rel]
   | po;[Rel];coi
   | [Acq];po;[R|W]
   | [Rel];po;[Acq]
   | [Xw];rfi;[Acq]
   | rmw
   | addr
   | data
   | data;coi
   | ctrl;[W]
   | ctrl;[W];coi
   | addr;po;[W]
   | (addr|data);rfi
   | coe
   | rfe
   | fre
   )+
```

Do the same with the DMB.LD.

```
...
only has a cycle if the following has one
   ([DMB.SY];po
   | po;[DMB.SY]
   | [R];po;[DMB.LD];po;[W|R]
   | [W];po;[DMB.ST];po;[W]
   | [R];(ctrl|(addr;po));[ISB];po;[W|R]
   | po;[Rel]
   | po;[Rel];coi
   | [Acq];po;[R|W]
   | [Rel];po;[Acq]
   | [Xw];rfi;[Acq]
   | rmw
   | addr
   | data
   | data;coi
   | ctrl;[W]
   | ctrl;[W];coi
   | addr;po;[W]
   | (addr|data);rfi
   | coe
   | rfe
   | fre
   )+
```

And do the same with the DMB.SY edges.

```
...
only has a cycle if the following has one
   (po;[DMB.SY];po
   | [R];po;[DMB.LD];po;[W|R]
   | [W];po;[DMB.ST];po;[W]
   | [R];(ctrl|(addr;po));[ISB];po;[W|R]
   | po;[Rel]
   | po;[Rel];coi
   | [Acq];po;[R|W]
   | [Rel];po;[Acq]
   | [Xw];rfi;[Acq]
   | rmw
```

```
   | addr
   | data
   | data;coi
   | ctrl;[W]
   | ctrl;[W];coi
   | addr;po;[W]
   | (addr|data);rfi
   | coe
   | rfe
   | fre
   )+
```

Rearrange.

```
...
= (rfe | fre | coe
   | addr | data
   | ctrl; [W]
   | (ctrl| (addr; po)); [ISB]; po; [W|R]
   | addr; po; [W]
   | (ctrl | data); coi
   | (addr | data); rfi
   | rmw
   | [Xw];rfi;[Acq]
   | po;[DMB.SY];po
   | [Rel];po;[Acq]
   | [R];po;[DMB.LD];po;[W|R]
   | [Acq];po;[R|W]
   | [W];po;[DMB.ST];po;[W]
   | po;[Rel]
   | po;[Rel];coi
   )+
```

(ctrl|(addr;po));[ISB];po;[W] is subsumed by ctrl;[W], addr;po;[W]

```
...
= (rfe | fre | coe
   | addr | data
   | ctrl; [W]
   | (ctrl| (addr; po)); [ISB]; po; [R]
   | addr; po; [W]
   | (ctrl | data); coi
   | (addr | data); rfi
   | rmw
   | [Xw];rfi;[Acq]
   | po;[DMB.SY];po
   | [Rel];po;[Acq]
   | [R];po;[DMB.LD];po;[W|R]
   | [Acq];po;[R|W]
   | [W];po;[DMB.ST];po;[W]
   | po;[Rel]
   | po;[Rel];coi
   )+
```

Apply Xw = range(rmw). And strengthen the DMB.LD and [Acq];po[R|W] edges.

```
...
```

```
⊆ (rfe | fre | coe
  | addr | data
  | ctrl; [W]
  | (ctrl | (addr; po)); [ISB]; po; [R]
  | addr; po; [W]
  | (ctrl | data); coi
  | (addr | data); rfi
  | rmw
  | [range(rmw)];rfi;[Acq]
  | po;[DMB.SY];po
  | [Rel];po;[Acq]
  | [R];po;[DMB.LD];po
  | [Acq];po
  | [W];po;[DMB.ST];po;[W]
  | po;[Rel]
  | po;[Rel];coi
  )+
```

Finally, the relation above is the same as ARMv8-axiomatic's ob with the definitions of obs, dob, aob, and bob inlined, and the transitive closure replacing the recursive definition.

□