A Formalization of Java's Concurrent Access Modes

ANONYMOUS AUTHOR(S)

Java's memory model was recently updated and expanded with new access modes. The accompanying documentation for these access modes is intended to make strong guarantees about program behavior that the Java compiler must enforce, yet the documentation is frequently unclear. This makes the intended program behavior ambiguous, impedes discussion of key design decisions, and makes it impossible to prove general properties about the semantics of the access modes.

In this paper we present the first formalization of Java's access modes. We have constructed an axiomatic model for all of the modes using the Herd modeling tool. This allows us to give precise answers to questions about the behavior of example programs, called litmus tests. We have validated our model using a large suite of litmus tests from existing research which helps to shed light on the relationship with other memory models. We have also modeled the semantics in Coq and proven several general theorems including a DRF guarantee, which says that if a program is properly synchronized then it will exhibit sequentially consistent behavior. Finally, we use our model to prove that the unusual design choice of a partial order among writes to the same location is unobservable in any program.

1 INTRODUCTION

The original Java memory model [Manson et al. 2005] included an early attempt to define the semantics of lock-free shared memory programs running on the Java platform, but the definitions were hard to understand and there was no easy way to check the behavior of example programs. It was also later discovered that it ruled out existing compiler optimizations which it claimed to support [Ševčík and Aspinall 2008]. Since then, researchers have made great advances in memory model design while studying other weak memory models like those for ARM [Alglave et al. 2008; Pulte et al. 2017], C11 [Batty et al. 2011; Kang et al. 2017; Lahav et al. 2017; Vafeiadis et al. 2015], Power [Alglave et al. 2014], and x86 [Owens et al. 2009].

Recently, the ninth version of the Java Development Kit updated and expanded Java's memory model using new "access modes". Though the design of the access modes is inspired by C11's memory orders [Committee et al. 2010], it differs in a few key ways. First, it sheds complicated legacy features like release sequences and release-consume accesses. Second, it includes a broad but simple mechanism to forbid so called "out of thin-air" behavior [Batty and Sewell 2014]. Finally, it makes no provision for a total order on writes to the same location. Taken together this suggests new opportunities to use a simpler model, develop metatheory, and verify lock-free algorithms for the Java platform.

However, the documentation [JDK9 2017; Lea 2017, 2018] is frequently ambiguous. This makes it extremely difficult to provide definitive answers about program behavior and there is little hope of proving important properties about the semantics. Further, it impedes the discussion of key features of the model's design.

To address these issues, we present the first formalization of Java's access modes. Critically, our model is *precise* and *complete*, formalizing all of the main features of Java's access modes. Additionally, we have endeavoured to make the model as *readable* as possible, defining all modes and fences as small extensions to an intuitive notion of visibility. Specifically, we make the following contributions:

- an axiomatic model for all of Java's access modes, fences, and atomic read-writes;

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

- an instantiation of our model for the Herd tool, which allowed us to examine the outcomes for more than 80 test programs; and
- an instantiation of our model in Coq, which allowed us to prove three key theorems about the semantics that are novel for Java's access modes.

We have constructed our model using the `cat` language. This allowed us to leverage the Herd tool [Alglave et al. 2014] to give definitive answers to questions about example programs, called litmus tests. Litmus tests are designed to highlight specific behaviors in memory models. Herd enumerates all possible executions for a litmus test and determines which are allowed according to the model. Then, if at least one execution is allowed, the behavior illustrated by the litmust test is allowed by the model.

We used Herd with more than 80 litmus tests drawn from prior research to help validate our model by comparing it to ARMv8, C11 and x86. Our goal for these comparisons is to show that there are no unexpected differences in behavior. For example, the conventional wisdom is that language memory models will exhibit more behaviors than architecture memory models due to aggressive compiler optimizations. Thus, Java's access modes should permit more behaviors than ARMv8. If that is not the case then it should be due to a deliberate design decision and not a bug in the definitions. For all 80 litmus tests our model behaves according to our expectations and the documented design of the access modes.

We have also formalized our model in Coq and we prove three key theorems: absence of causal cycles when all reads are "release" reads, sequentially consistent semantics under proper synchronization (DRF), and a guarantee that each stronger mode admits fewer executions [Vafeiadis et al. 2015]. These theorems further validate the definitions of our model, give more evidence that the model is complete with respect to the documentation, and clearly demonstrate that the semantics is suitable for formal reasoning.

Finally, the partial order on same-location writes in Java's access modes represents a significant departure from the conventions of existing memory models. We show that the impact of switching to a partial order is "unobservable" in any example program executed using our model and thereby show that Java's access modes can adopt a total order on writes to the same location. The simplicity of our model shines through in our proof, which makes it clear that our reasoning is not applicable to the more complex axiomatic models of RC11 and ARMv8.

The rest of this paper is laid out as follows. In Section 2 we use a simple example to illustrate the ambiguity of the documentation. In Section 3 we detail the features which distinguish Java's access modes from other memory models. In Section 4 we give a complete account of our axiomatic model. In Section 5 we detail the results of our comparisons with ARMv8, C11 and x86. In Section 6 we give a formalization of the model in Coq and use it to prove the theorems listed above. In Section 7 we demonstrate that the choice of a partial coherence order is not observable in our model. In Section 8 we discuss related work.

2 JAVA'S ACCESS MODES

Here we will introduce the basics of the Java access mode API and motivate our formalization by way of an example program. We will show how different interpretations of the documentation can cause one of three unwanted outcomes when using the access mode API.

Developers can make use of Java's Access Modes, hereafter "the JAM", through the `VarHandle` API included in the JDK version 9. There are four access modes: plain, opaque, release-acquire, and volatile. Regular reads and writes of shared variables are considered plain mode, and the `VarHandle` API allows reads and writes to be annotated with one of the other three modes. The specified intent of the JAM is that each mode provides progressively more guarantees about the behavior of its accesses at the expense of some performance.

$$\text{plain} \sqsubseteq \text{opaque} \sqsubseteq \text{release-aquire} \sqsubseteq \text{volatile}$$

Plain mode gives virtually no guarantees and the compiler is allowed free reign in optimizing such memory accesses. On the other hand volatile mode provides sequentially consistent (SC) semantics when it is used for all accesses, but requires a memory barrier for each access which can make execution slow [1].

The `VarHandle` documentation [JDK9 2017; Lea 2017, 2018] is intended to answer questions about which access mode should be used for a read or write to maximize performance while providing enough guarantees to ensure program correctnes. It cites three bad outcomes arising from the confusion around the old Java memory model as motiviation for the new access modes: undersynchronized and broken code, oversynchronized and slow code, and platform specific code. However, the documentation is written in loose prose which can result in any one of the same three outcomes which the access modes were created to address.

To see this, consider the example execution of a message passing program in Figure 1. An execution graph like this catalogues the relationships and effects of each memory access from and example program. $\mathtt{M}(x, n)$ represents a memory access of x with the value n where `M` can be write, `W`, a read, `R` and later an atomic read-write `RW`. Relationships between memory access are represented as directed edges. The dashed edge labeled with `rf` represent reads-from and the edges labeled with po represent the sequence of instructions as defined in the program. Note that `a` and `b` are concurrent with `c` and `d`.

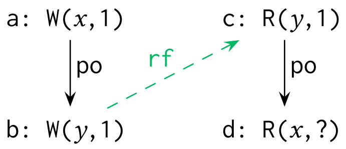


Fig. 1. Event Graph

Intuitively, if the read of y produces the value 1, it acts as a signal to the rest of the second thread that the write to x has completed. However, in the presence of weak memory accesses the sequence of execution may not follow program order. Thus, an important question is: which access modes should one use so that `d` reads the value 1 from `a` in keeping with program order, and not 0 from the initialization of x? Depending on a reasonable interpretation of the documentation, our answer to this question may result in one of the aforementioned bad outcomes.

Undersynchronized, broken code. Consulting the documentation [Lea 2018] we find that opaque mode makes the following guarantee:

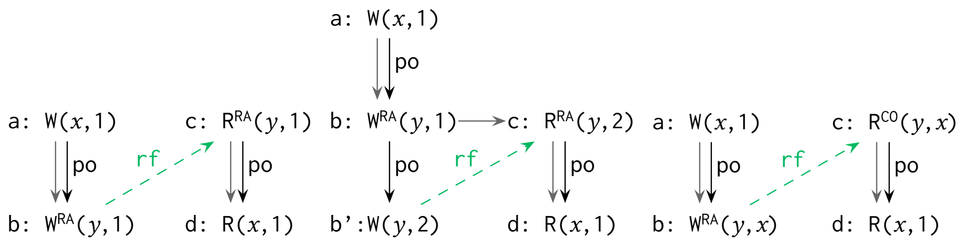
> If Opaque (or any stronger) mode is used for all accesses to a variable, updates do not appear out of order.

Whether opaque mode is the correct mode for each access to ensure that `d` reads 1, depends heavily on the definition of “out of order”. Intuitively, we assume that `rf` implies that the read has observed the effects of the write. Then, if the “order” from the documentation for opaque mode is program order, we might reasonably conclude that updates (writes) are observed in program order. In which case, `d` observes the effects of `c` and `b` observes the effects of `a`. Then, through the `rf` edge we can conclude that `d` observes the effects of `a` and `d` will read 1.

Unfortunately this approach actually leaves the code undersynchronized and it will not work as described. It turns out, the documentation also says that opaque mode makes no guarantees about information that travels through reads:

> ...reading a value in Opaque mode need not tell you anything about values of any other variables.

[1]This property, of progressively greater guarantees, is called monotonicity and we prove it as a theorem for our model in Section 6



(a) Message Passing, Fixed (b) Release-sequence (c) Consume-read

Fig. 2. Release sequences and Consume-reads

Thus, whether or not writes are observed in program order, the read c may not tell us anything about what happened with a opening the way for d to read 0 from the initial write to x. These two pieces of documentation can lead to an inconsistent understanding of opaque mode. A reasonable reader may react defensively and look to the stronger modes to guarantee the correct program behavior.

Oversynchronized, slow code. A defensive approach to getting the right program behavior is to employ volatile mode. This would give all of the reads and writes in the program SC semantics, guaranteeing that everything is observed in program order. Thus, the po and rf edges would indeed result in d observing a, and d will read 1. However, if the compiler takes the approach outlined in the documentation it will insert a full fence after every SC access which will slow down execution.

Platform specific implementation. Alternately, we may note that x86 processors do not reorder writes with writes or reads with reads. Assuming opaque mode accesses will translate directly to hardware instructions, we can tag each access as opaque and use same reasoning that we used for volatile mode. Unfortunately this reasoning is unsound for execution on ARM processors, which allow such reorderings, and the code would not be portable.

As it happens, the best solution is to use release-write for b and acquire-read for c. These modifications appear as the RA annotations for the write and read of y in Figure 2a. This will ensure the orderings, depicted as gray edges, for any access before b and any access after c with a minimum of synchronization. Thus, a is observed by d through those orderings and d will read the value 1. Critically, we can only demonstrate this after more precisely defining the guarantees provided by the opaque and release-acquire modes of the JAM. Our formalization brings this much needed clarity to the current specification.

3 DISTINGUISHING FEATURES OF THE JAM

While the JAM was inspired by the access modes of C11, it makes several departures from C11 and other memory models that are worthy of consideration and a challenge for formalization. First, it sheds legacy features like the release sequences and consume-reads of C11. Second, it contains a broad and simple definition of causal cycles for the purposes of ruling out thin-air reads. Finally it includes a non-total ordering on writes to a particular memory address, which is called the coherence order. Here we will discuss how each of these differences will impact any formalization effort for the JAM.

3.1 Letting Go of Release Sequences

Release sequences and consume-reads can be seen as specialized variants of the release-acquire memory order in C11 and release-acquire mode in the JAM. The idea is that, in certain cases, it's possible to get the same guarantees as a release-write and acquire-read pair, like the one Figure 2a, with less synchronization. The result is faster execution in some contexts.

```
r1 := x  ||  r2 := y
y  := r1 ||  x  := r2
```

(a) Out of Thin-air

$$\begin{tikzcd}
\mathtt{a:\ R}(x,\mathtt{n}) \arrow[d, "\mathtt{po}"'] & \mathtt{c:\ R}(y,\mathtt{n}) \arrow[d, "\mathtt{po}"] \\
\mathtt{b:\ W}(y,\mathtt{n}) \arrow[ur, dashed, "\mathtt{rf}"'] & \mathtt{d:\ W}(x,\mathtt{n}) \arrow[ul, dashed]
\end{tikzcd}$$

(b) Causal Cycle

```
r1 := x  ||  r2 := y
y  := 1  ||  x  := 1
```

(c) Load-buffering

Fig. 3. Causal Cycles

Release sequences can be used when a write that is after a release-write is read by an acquire-read. Consider the message passing variant in 2b where b is followed by another distinct write to y, b'. If c reads from b' then the release sequences of C11 would guarantee that d would see a through the orderings depicted with gray edges. Then d will read the value 1, just as it would if c read from b. Without the guarantees of release sequences, b' must also be a release-write to get the desired outcome.

Consume-reads are used with a release-write when the memory accesses which must be ordered after the consume-read are data dependent on the value of the read. Consider the message passing variant in 2c where c is annotated with CO and reads a pointer that determines the memory address that d reads from. Some architectures enforce the ordering of c and d in the presence of such data dependencies without the extra synchronization that would result from making c an acquire-read. This can speed up execution in those settings.

The JAM does not include the guarantees of release sequences or any way to annotate a read as a consume-read. This is a design choice in favor of simplicity in the model and it gives us the opportunity to build a more readable and more easily testable model.

3.2 Acyclic Causality

In Figure 3a we have a classic example program which can exhibit a so called “thin-air” read under sufficiently weak memory models. The question is, at the end of execution can $x = y = 42$? Intuitively, assuming both x and y are initialized to 0, this program can’t generate 42 “out of thin-air”, but many axiomatic models do not exclude such executions from the set of all candidate executions.

Observe that, in any execution that allows $x = y$, there must be a cycle in the program order and reads-from relations, as illustrated in Figure 3b. The JAM explicitly forbids such cycles, but at the cost of forbidding some behaviors which may be beneficial for performance.

For example, consider the classic load-buffering (LB) litmus test in Figure 3c, where the question is, can r1 = r2 = 1? If performance was the sole concern in the design of the JAM this behavior would be allowed because the reads can be reordered with the writes with the aim of improving performance. Unfortunately, this example also exhibits the same cycle in the program order and reads-from, so it is forbidden by the JAM.

The problem of differentiating these kinds of examples has been studied at great length by memory model researchers. Another approach is to forbid cycles in rf and a subset of program order based on a notion of dependency. Sadly, this too has very subtle issues, as outlined by Batty and Sewell [2014]. The original, formal Java Memory Model [Manson et al. 2005] attempted to address the issue of causal cycles in its full generality. More recently the Promising semantics of [Kang et al. 2017] introduced a novel "promise" mechanism to model compiler optimizations for this purpose.

In all of these cases the complexity of the resulting models makes them hard to understand and hard to test. Instead, the JAM specification adopts a simple solution by forbidding cycles in the program order and reads-from. While this does forbid the behavior of the second example at the cost of some performance [Ou and Demsky 2018], it gives us yet another opportunity to build a simpler model.

3.3 Partial Coherence Order

The JAM specification makes no provision for a total ordering of writes to a given memory location which is a standard feature in other memory models. The consequences of this design choice manifest in subtle ways.

For example, the standard definition of atomicity for read-writes relies on a total coherence order. Borrowing the definition from Vafeiadis et al. [2015], for a read-write, RW, the write it reads-from, $\mathtt{W}_1$, and the coherence order, $\xrightarrow{co}$, we have:

$$\mathtt{W}_1 \xrightarrow{rf} \mathtt{RW} \implies \neg\exists\, \mathtt{W}_2, \mathtt{W}_1 \xrightarrow{co} \mathtt{W}_2 \xrightarrow{co} \mathtt{RW}$$

Taken together with a total coherence order this means that the atomic pairs of writes and read-writes are always totally ordered. Since the JAM makes no such guarantees regarding normal writes there are ambiguities like the one in Figure 4. There, the write-read-write pairs are not ordered across threads since there is no global relationship between writes.

As a consequence, our model must include extra contraints for read-writes while being careful not to over-constrain normal writes which would otherwise be concurrent. We detail our approach in Section 4.5.

Separately, our drive for simplicity in the definitions of the model has yielded a key insight where the coherence order is concerned. While we have remained faithful to the documentation and personal communications with the authors in modeling a partial coherence order, we will show that the effects of that choice are not observable under our model. Specifically, we show that it is impossible to construct a litmus test that behaves differently in the presence or absence of a total coherence order.

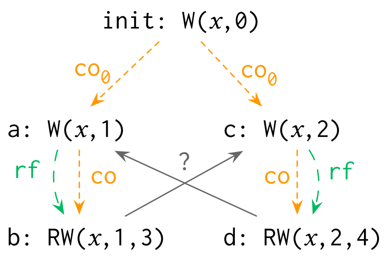


Fig. 4. Concurrent Read-writes

4 AXIOMATIC MODEL

The JAM has six components: plain mode, opaque mode, release-acquire mode, volatile mode, fences, and atomic read-writes. Each of the modes, from plain to volatile, provides strictly more guarantees than the previous mode.

To model the JAM we have constructed an axiomatic semantics. The definitions of our model act as a predicate over candidate executions which include memory events, e.g. reads and writes, and relations over those events, e.g. reads-from and program order, in the style of Alglave et al. [2014].

Our definitions focus on two key concepts. The first is an acyclicity requirement for the coherence order. Aside from the restriction of causal cycles, this is the only mechanism by which executions are forbidden. Thus, every unwanted execution must exhibit a cycle in the coherence order. The second is an intuitive notion of *visibility*, which represents when one memory access has "seen" the effects of another memory access. The behavior of the three modes and fences are modeled as small extensions to this relationship.

In each of the following subsections we detail the extensions to our model for each component. The full model can be viewed in Appendix A.

4.1 Plain & Opaque Mode

In the JAM documentation, plain mode accesses are given virtually no guarantees when they occur in different threads without correct synchronization. Opaque mode, on the other hand, does provide some cross thread guarantees which form the basis for the rest of the memory model. There are some subtleties involved in the documented relationship between plain and opaque mode accesses so we will address them together. First, the main properties that opaque mode accesses guarantee are:

Bitwise Atomicity Reads will see the value of only one write. Opaque mode guarantees that reads will not see mixed bits from different writes.

Write Availability Writes can be read by later reads. The intent is to avoid a situation (e.g. in a spin-loop) where repeated reads never see a write in another thread because they are optimized by the compiler to execute only one time. When an optimization like this happens, the availability of the write for the read depends on when the read is executed [Corbet 2012].

Acyclic Causality A read should not influence its own value. As described in Section 3.2, this forbids counter-intuitive behavior like thin-air reads.

Coherence The order of writes should respect visibility and should agree with the way that reads observe their order. For example, one guarantee (of four we will define later) is that a read should be paired with the last write that it knows about and not an earlier one.

Our model of opaque mode focuses on acyclic causality and coherence. In keeping with other axiomatic models built for Herd [Alglave et al. 2008; Lahav et al. 2017; Pulte et al. 2017], our model pairs each read with a single write and there is no accounting for optimizing reads out of loops as Herd does not support them.

Plain Coherence. The JAM documentation does not include a coherence guarantee for plain mode accesses. This follows the approach in the C11 documentation but it departs from formal models for C11.

To see why plain mode accesses should be included in the coherence ordering guarantees, note that plain mode accesses are safe to use within a critical section guarded by a lock according to the documentation. The idea is that code which is properly synchronized with locks will have single threaded semantics for the duration of the critical section. Thus, the model shouldn't require accesses to be annotated with anything stronger than plain mode.

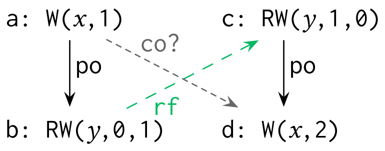

$$\begin{tikzcd}
\texttt{a: W}(x,\texttt{1}) \arrow[d, "\texttt{po}"] \arrow[dr, dashed, "\texttt{co?}"] & \texttt{c: RW}(y,\texttt{1},\texttt{0}) \arrow[d, "\texttt{po}"] \\
\texttt{b: RW}(y,\texttt{0},\texttt{1}) \arrow[ur, dashed, "\texttt{rf}"'] & \texttt{d: W}(x,\texttt{2})
\end{tikzcd}$$

Fig. 5. Message Passing Coherence

Then, consider the message passing variant in Figure 5. Here, the purpose of this pattern is to signal the end of the critical section through atomic read-writes that unlock, b, and lock, c, the variable y. For any lock to work correctly, the accesses which are program order before the unlock, a $\xrightarrow{\text{po}}$ b, must be visible to accesses after the lock, c $\xrightarrow{\text{po}}$ d (for now we leave the mechanism that enforces these orderings unspecified). In particular the effects of the write a should be visibile to the write d, a $\xrightarrow{\text{co?}}$ d.

However, even if the unlock and lock enforce program order of the plain writes, and thereby show that a is visible to d, we would not be able to derive a $\xrightarrow{\text{co}}$ d because the coherence rules do not apply to plain accesses. This stands in contrast to formal models of C11 [Lahav et al. 2017] where the coherence order and happens-before relations apply to plain accesses. As a result, our model extends the coherence order guarantees to plain mode accesses and only the extra guarantee of acyclic causality is left to opaque mode.

Herd Model. Figure 6 details our axiomatic model of the JAM's plain and opaque modes as defined in the `cat` modeling language. `cat` includes the following built in operations: `|`, `&`, `;`, `+`, `~` are relational union, intersection, composition, transitive closure, and complement. New constructs are defined with `let`. Filters, like wwco, are defined on relations using `let F(R) =`... and applied with `F(R)`. Finally, models can include checks like `acyclic` for relations.

Herd provides several built-in, ambient sets and relations for models defined in cat. IW, W, FW, R, O, RA, V are the sets of initial writes for each location, plain mode writes, the final plain mode writes to each location, plain mode reads, opaque accesses, release-acquire accesses, and volatile accesses. po, po-loc, rf, and invf represent the program order, program order per-location, reads-from, and inverted reads-from relations. loc relates memory accesses to the same location and id is the identity relation. Note, that we do not use Herd's built-in coherence relation but rather define our own without a total ordering.

```
let vo = rf+ | po-loc

let wwco(rel) = rel & ~id
                & loc & (W * W)

let coww = wwco(vo)
let cowr = wwco(vo;invrf)
let corw = wwco(vo;po-loc)
let corr = wwco(rf;po-loc;invrf)

let coinit = wwco((IW * W))
let cofw = wwco((W * FW))

let co = coww | cowr | corw
       | corr | cofw | coinit

acyclic co

let opq = O | RA | V
acyclic (po | rf) & opq
```

Fig. 6. Opaque Mode

We define visibility order, vo, using two properties, po-loc and rf+. In the first case, given two memory accesses to the same location in the same thread the second should see the effects of the first. In the second case, the sequence of one or more reads guarantees that the final read is executing in a context where the effects of the write are visible. This definition for vo guarantees only the most basic forms of ordering which is important given the inclusion of plain mode accesses.

Then, to satisfy the coherence requirement we first ensure that the coherence order includes only distinct writes to the same location, wwco, and adopt the four behaviors originally identified for C11 by Batty et al. [2011]:

coww Two such writes ordered by visibility are similarly coherence ordered.
cowr A read chooses the last write it has seen.
corw A write that follows a read of the same location in program order is coherence ordered after the write paired with the read.
corr Program ordered reads to the same location order their writes in the same way.

We also order the initialization write before all writes of the same location, coinit, and we order all writes before the final write of the same location, cofw. In the rest of the paper these relationships will be distinguished as co_0 for clarity but they are treated the same as any other co edge by the model.

These six rules make up the definition of the coherence order, co, until we discuss read-writes. Then, the primary mechanism by which the model forbids executions is through requiring the coherence order to be acyclic, acyclic co. In Figure 7, we give examples of how each coherence rule forbids an execution by showing a cycle in the coherence order. In every example, the final write of 1 is assumed to be coherence order after all other writes, co_0.

In Figure 7a, if we the write, a, is visible write, b, to the same location then a is not the last write. In Figure 7b, if the read c has seen the last write b then it should not be able to read an older write a. In Figure 7c, if we can follow the read of a to b to a later write, c, then a can't be the last write. In Figure 7d, given two reads in program order, b and c, if b has seen the last write then c should not be able to read an older write.

Finally, we must forbid causal cycles for opaque mode access. We define opq to be any opaque or stronger access since any guarantee provided by opaque should apply to stronger access modes. Then, we forbid cycles in the union of program order and reads-from, qualified for opaque mode accesses: acyclic (po | rf) & opq.

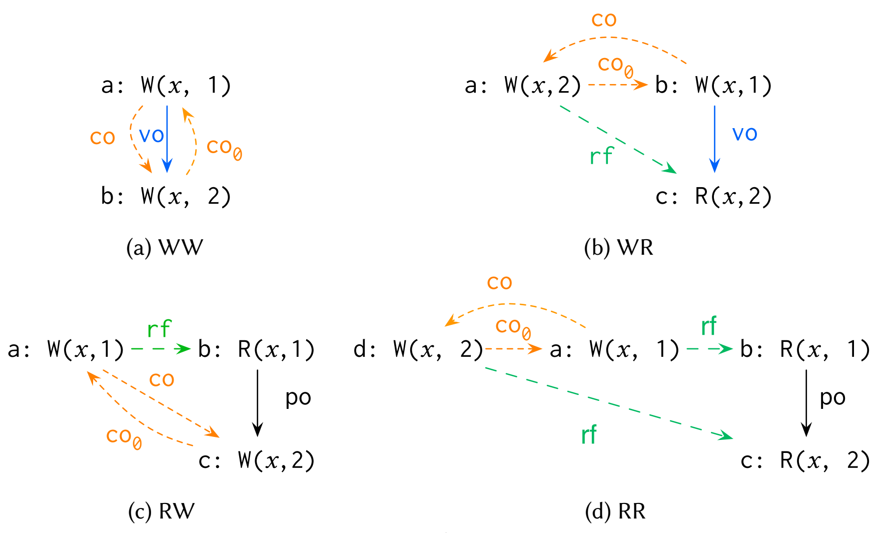


Fig. 7. Coherence

As we will see in the following sections, these base definitions allow us to model nearly every other component by extending vo. The lone exception is atomic read-writes, for which we extend the coherence order directly.

4.2 Fences

The JAM supports five types of fences: release, acquire, load-load, store-store, and full. Programs often include fences to enforce the order of memory access effects before and after the fence. We update vo by extending visibility to rfso+ and abstracting over these fence types with *specified orders* [Bender et al. 2015; Crary and Sullivan 2015]. We therefore require that programs relate the ordering of pairs of memory accesses using either svo or spush in place of fences.

```
with to from linearisations(M\IW, cofw | rf | into)
let spushto = to+ & (domain(spush) * domain(spush))
let rfso    = rf | svo | spush | spushto;spush
let vo      = rfso+ | po-loc
```

The effects of intra-thread ordering fences (release, acquire, load-load, and store-store) are modeled with specified visibility orders, svo. Full fences, are modeled as specified push orders, spush and spushto, in the style of Crary and Sullivan [2015].

This approach allows us to extend visibility to account for synchronization in a way that is uniform for fences and their related access modes. Moreover, the abstraction allows the compiler to make smart choices to ignore specified visibility based on existing intra-thread ordering of instructions guaranteed by a target architecture as discussed in the work of Bender et al. [2015]; Sullivan [2015]. We will see an example of this when discussing atomic read-writes.

Note that trace order, to, is a total ordering of all memory accesses (except for initial writes) as constructed by Herd's built-in linearisations function. Trace order respects cofw, rf, and all intra-thread orderings, into, induced by specified visibility orders, push orders and later release-acquire memory accesses and volatile memory accesses. Intuitively, each of these synchronization mechanisms guarantees that the related memory accesses are executed in program order.

Specified Visibility Orders. To see how specified visibility orders create intra-thread visibility between accesses, consider the event graph in Figure 8 for the message passing example from Section 2.

By specifying that $\mathsf{a} \xrightarrow{\mathsf{svo}} \mathsf{b}$ and $\mathsf{c} \xrightarrow{\mathsf{svo}} \mathsf{d}$ we can show that d could not have read 0 from the initial write to x. First, we assume that the initial write to x is coherence order before a, $\mathtt{init} \xrightarrow{\mathsf{co}_0} \mathsf{a}$. Then, assume that d has read from the initialization, $\mathsf{R}(x, 0)$. We will show a contradiction. In the graph we have that, $\mathsf{a} \xrightarrow{\mathsf{svo}} \mathsf{b} \xrightarrow{\mathsf{rf}} \mathsf{c} \xrightarrow{\mathsf{svo}} \mathsf{d}$. By the definition of `rfso` we have the three edges that combine to show $\mathsf{a} \xrightarrow{\mathsf{vo}} \mathsf{d}$. Then, by `cowr` (Fig. 7b) we have that $\mathsf{a} \xrightarrow{\mathsf{co}} \mathtt{init}$ which is a cycle in co and a contradiction.

Specified Push Orders. Specified push orders create visibility relationships in two ways. The first is an intra-thread visibility ordering between the two push ordered instructions like svo. This appears as the spush in the definition of `rfso`.

The second, is a cross-thread ordering that emulates the standard total ordering of two full fences. Given two push orders, the head of the first will be visible to tail of the second, or vice-versa based on the current ordering of the heads in to. The ordering of the heads is recorded as spushto which we connect with the additional visibility ordering of one tail using, spushto;spush.

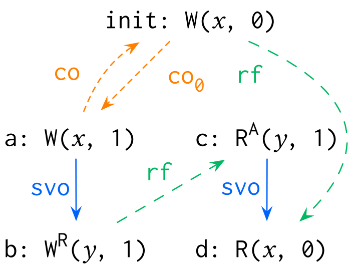


Fig. 8. Specified Visibility

To see how push orders emulate full fences, consider the store-buffering litmus test in Figure 9. The question is, can both reads take their values from the initialization?

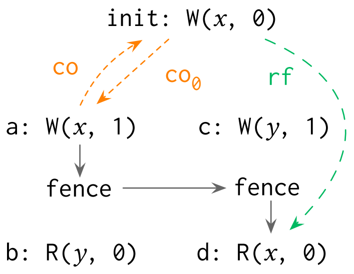


Fig. 9. Fences, One side

First, we assume the gray edge between the two `fence` instructions, `fence` $\longrightarrow$ `fence`, which represents one side of the total ordering provided by full fences. Then, we assume that d reads from the initial write to x and show a contradiction. Since fences also provide the intra-thread orderings $\mathsf{a} \longrightarrow$ `fence` and `fence` $\longrightarrow \mathsf{d}$, we can see that a is ordered before d. Then by `cowr` it must be that $\mathsf{a} \xrightarrow{\mathsf{co}} \mathtt{init}$, which creates a cycle with $\mathtt{init} \xrightarrow{\mathsf{co}_0} \mathsf{a}$ and a contradiction. On the other side of the ordering between fences a similar argument applies if b reads from initial write to y. Thus, b and c could not have read from the initial writes to x and y in the same execution.

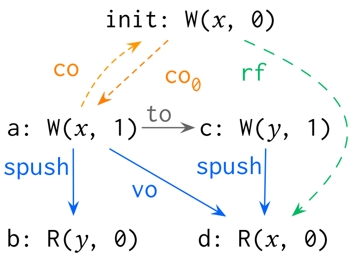


Fig. 10. Push order, One side

Push orders are more direct but capture the same ordering. In Figure 10 both write-read pairs are push ordered.

First we assume one side of the total trace order between the two writes, $\mathsf{a} \xrightarrow{\mathsf{to}} \mathsf{c}$. Again, we assume that d reads from the initial write to x and show a contradiction. By spushto;spush, we have $\mathsf{a} \xrightarrow{\mathsf{vo}} \mathsf{d}$ which means that d has seen a and the same reasoning with `cowr` that we used with the fences applies.

Note that including spushto in vo in place of spushto;spush would give the desired visibility between the heads and tails of two orders by transitivity, but it would also give an ordering to the heads of the push orders which does not otherwise exist.

Specified orders have a natural implementation as fences which has been studied by Bender et al. [2015] and Sullivan [2015]. We give such a mapping in Appendix B.

4.3 Release-Acquire Mode

We can now make small extensions to incorporate other access modes into our model. First, we define release-acquire mode following the standard set by other models like C11 [Committee et al. 2010] and ARMv8 [Pulte et al. 2017].

```
let rel = W & (RA | V)
let acq = R & (RA | V)
let ra  = po;[rel] | [acq];po | rfso
let vo = ra+ | po-loc
```

We define release writes, `rel`, to be any write marked as release or volatile. We define acquire reads, `acq` to be any read marked as acquire or volatile. Again, any guarantee provided by release-acquire mode should hold for volatile mode.

We update the `vo` definition from opaque mode with fences by extending what was previously `rfso+` to be `ra`. We add edges from memory accesses for any location to a release write that is later in program order, `po;[rel]`. We also add edges from an acquire read to program order later opaque memory accesses for any location, `[acq];po`. Note that the documentation makes clear that plain accesses should be ordered by release writes and acquire reads so long as the types are bitwise atomic.

To see, how the release-aquire extension to opaque mode works, consider the message passing example from the Section 2. Note that, we use superscript RA for release writes, $\mathtt{W}^{\mathtt{RA}}$, and acquire reads, $\mathtt{R}^{\mathtt{RA}}$. Later we will use V for volatile accesses. Then, if we adopt a release write for b and an acquire read for c we can show that, $\mathtt{a} \xrightarrow{\mathtt{vo}} \mathtt{d}$, and the reasoning is the same as for specified visibility orders in Section 4.2.

We note that our definitions suggest that, if all reads were release reads, we could derive a visibility relationship between a read and itself in executions exhibiting causal cycles. In section 6, we prove that visibility cycles are a contradiction in our model and, as such, causal cycles are forbidden when all reads are release-reads.

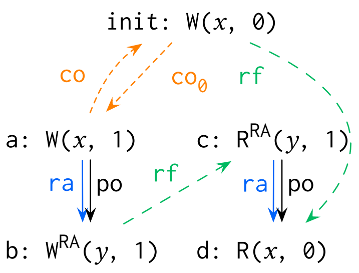


Fig. 11. Release-Acquire Visibility

4.4 Volatile Mode

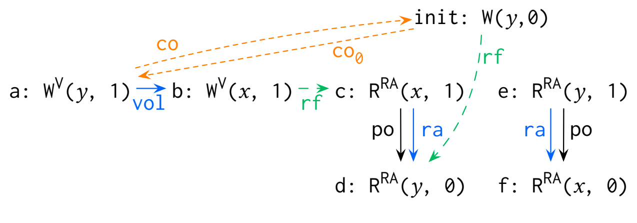
Volatile mode is a further extension of the visibility order in release-acquire mode. We update `vo` by extending `ra` to `vvo`. Here `volint` creates edges from any access to a volatile read that is later in program order, `po;[V & R]`, and from an volatile write to any program order later opaque memory access, `[V & W];po`.

```
let vol = V
let volint = po;[vol & R] | [vol & W];po
let push = spush | volint
let vvo = ra | pushto;push | push
let vo = vvo+ | po-loc
```

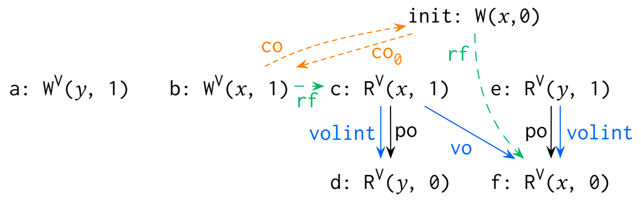
These new edges preserve program order for any access before or after the volatile access when combined with the visibility definition of release-acquire. We also extend `spush` with `volint` edges so that we can leverage the same visibility relationships induced by push orders for volatile accesses.

A simpler approach would be to translate the total trace ordering into visbility edges. That is, we could replace the definition of `vol` above with the following:

```
let vol = ra | spush | volint | to & (V * V)
```



(a) IRIW Release-Acquire



(b) IRIW Volatile

Fig. 12. IRIW Variants

This approach more directly encodes the cross thread ordering guarantee of sequential consistency, but, unfortunately, this is too strong.

Consider a modified version of the classic IRIW litmus test in Figure 12a. The question is, can both d and f read from the initial write to x and y? As outlined in the JAM documentation the acquire reads in this example are allowed to see the writes in different orders, so both can read from the initial writes for x and y. However, if volatile mode accesses are totally ordered as in the proposed definition, then we have either a $\xrightarrow{\text{vo}}$ b or b $\xrightarrow{\text{vo}}$ a. In Figure 12a we have the first case. Then we have a $\xrightarrow{\text{vo}}$ d+ because a $\xrightarrow{\text{vol}}$ b $\xrightarrow{\text{rf}}$ c $\xrightarrow{\text{ra}}$ d. Then by cowr, we have that a $\xrightarrow{\text{co}}$ init, which is a contradiction. In the other case, b $\xrightarrow{\text{vo}}$ a, we have a contradiction when f reads from the initial write to y.

Instead, we extend our notion of push orders and define push to include both spush and volint edges. This has the same effect as push ordering accesses related by volint. As we will demonstrate later, using a matching litmus tests, this guarantees the correct behavior for the release-acquire variant of IRIW because it enforces no direct oridering between the two writes.

Importantly, it also gives the correct behavior when the reads are volatile, which should be SC semantics. Consider Figure 12b which has one of the two possible orderings given by pushto;push when the reads are volatile. This correctly establishes c $\xrightarrow{\text{vo}}$ f creating a contradiction for the opposite thread's final read. As before the other direction of the total order forbids the other read.

4.5 Atomic Read-Writes

The behavior of atomic read-writes is the only part of the JAM that is not modeled by extending vo. Recall Figure 4 from Section 3. We must take care to ensure that we do not allow concurrent read-writes in the presence of a non-total coherence order. To achieve this we update the coherence order co with two additional rules:

```
let corwexcl = wwco((rf;[RW])^-1;co')
let corwtotal = wwco(((RW * W) | (W * RW)) & to)
let rec co = ... | crwexcl | corwtotal
```

The first, corwexcl ensures exclusivity in the relationship between the read-write and its paired write. Note that, for clarity, we separate out corwexcl even though it recursively refers to co in its definition. As illustrated in Figure 13, if there is a write co-after the one paired with the read-write, a $\xrightarrow{\text{co}}$ b then it is co-after the read-write, c $\xrightarrow{\text{co}}$ b.

Recalling the example of Figure 4, this exclusivity is not enough to prevent concurrent read-read-write pairs to the same location. Since exclusivity uses the ordering of other writes with the paired write we could consider a total ordering just for paired writes.

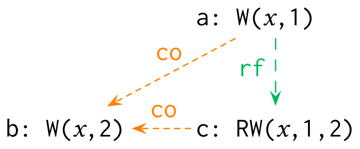


Fig. 13. Read-write Exclusivity

```
let pairedw = domain(rf;[RW])
let corwtotal = wwco((pairedw * pairedw) & to)
```

This suffices to forbid concurrent read-write chains to the same location. In Figure 4, by `corwtotal` we have a $\xrightarrow{co}$ c or c $\xrightarrow{co}$ a. Then, in the first case, we have b $\xrightarrow{co}$ c by `corwexcl`. The other side is similar. However, this approach inadvertently orders writes that could be concurrent under a partial coherence order. Instead, as in the first definition, we choose a total ordering with the read-write itself.

Now any chain of read-writes will be exclusive without unnecesesarily ordering regular writes. We will show that in Figure 4 either, b $\xrightarrow{co}$ c or d $\xrightarrow{co}$ a. By `corwtotal` we have d $\xrightarrow{co}$ a or a $\xrightarrow{co}$ d. In the first case we are done. In the second case, we consult `corwtotal` again and we have b $\xrightarrow{co}$ c or c $\xrightarrow{co}$ b. In the first case we are done. In the second case, by assumption we have a $\xrightarrow{co}$ d and c $\xrightarrow{co}$ b. Then we can apply `corwexcl` to both to derive c $\xrightarrow{co}$ d and d $\xrightarrow{co}$ c, which is cycle and a contradiction. Importantly this reasoning can be applied repeatedly to any chain of read writes to achieve exclusivity.

We note here, that JAM makes no intra-thread ordering guarantees for atomic read-writes even though they exist on some architectures like x86 [Owens et al. 2009]. We discuss this in more detail in Appendix B.

4.6 Summary

Our axiomatic model of the JAM is complete, covering all four modes, atomic read-writes, and all five fence types in less than 40 lines of definitions. Moreover each component is implemented as a modest extension to just two relations which makes it readable. Now we must demonstrate that the model is consistent with expectations about the behavior of the JAM as outlined in the documentation. We discuss the results of our comparison with ARMv8, RC11 and x86 in the next section.

5 VALIDATION

In this section we validate our formalization of the JAM by comparing litmus test outcomes for the JAM with the outcomes for ARMv8 [Pulte et al. 2017], RC11 [Lahav et al. 2017] and the model for x86 included with Herd. We use these comparisons to show that there are no unexpected differences in behavior between each pair of models. For example, we expect the JAM to permit more behaviors than ARMv8 with the exception of litmus tests like load-buffering (Fig. 3c) because the behavior is forbidden by the JAM's definition of causal cycles.

We run the tests using the Herd tool [Alglave et al. 2014]. Herd exhaustively enumerates all possible executions of a litmus test and checks if each execution is allowed by the model being tested. We can then compare the three possible results for each model to see how often executions are allowed: `Always`, `Sometimes`, and `Never`. In terms of the behavior being tested `Always` and `Sometimes` mean that the behavior is allowed, while `Never` means that it is not allowed. Weaker memory models should allow more behaviors and see more `Always` and `Sometimes` results, while stronger memory models should allow fewer behaviors and see more `Never` results.

Our test suite is built with litmus tests taken from existing research for the three models we compare against. Importantly we did not modify any of these tests. However, Herd uses different built-in relations to refer to the access types of each model. For example, Herd provides the `L` built-in

Pulte et al. [2017]

name	ARMv8	JAM
WRC+addrs	Never	Never
LB+data+data-wsi	Never	Never
W+RR	Never	Never
totalco	Never	Never
PPOCA	Sometimes	Sometimes
IRIW	Sometimes	Sometimes
IRIW+addrs	Never	Sometimes
IRIW+poaas+LL	Never	Sometimes
IRIW+poaps+LL	Never	Sometimes
MP+dmb.sy+addr-ws-rf-addr	Never	Sometimes
WW+RR+WW+RR+wsilp+poaa+wsilp+poaa	Never	Sometimes
LB	**Sometimes**	**Never**

Fig. 14. ARMv8 Litmus Test Comparison

to refer to release-writes in ARMv8 litmus tests and the REL built-in to refer to release-writes in C11 litmus tests. Thus, for each comparison we include a mapping between the access types of the other model and the access modes of the JAM.

For each comparison we define the mapping between the built-in relations of the two models, we detail our expectations for the results, and then discuss the results of the comparison. As we will see our model behaves as expected in all cases.

5.1 Comparison with ARMv8

Herd provides several built-in relations for the ARMv8 litmus tests: M for normally memory operations, L for release writes, A for acquire reads, and DMB.SY for full fences. There are no SC/Volatile accesses.

```
let opq = M
let rel = L
let acq = A
let spush = po;[DMB.SY];po
```

Fig. 15. ARMv8 Mapping

Mapping. We map every memory access to opaque mode with `opq = M`. This is conservative with respect to our expectation that the JAM permits more behaviors than ARMv8. If we were to map some accesses to plain mode they would be allowed to exhibit causal cycles. Thus, mapping all memory accesses to opaque mode means the JAM model will permit fewer behaviors.

We map release-writes to release-acquire mode writes, `rel = L` and acquire-reads to release-acquire mode reads, `acq = A`. Note, that M includes L and A so release writes and acquire reads will also be treated as opaque mode accesses. Finally we treat all accesses with a full fence between them in program order having a specified push order.

Expectations. As stated, we expect the JAM to be weaker than ARMv8 since the JAM is a language memory model which is subject to aggressive compiler optimizations. That is, we expect that any time ARMv8 exhibits a behavior the JAM should too and there should be instances where the JAM exhibits behaviors that ARMv8 doesn't. The lone exception is cases where the broad definition of causal cycles adopted by the JAM will rules out behavior like the load-buffering example of Figure 3c.

Results. Aside from `totalco` and LB, all the tests come from the supplementary material accompanying the ARMv8 model of Pulte et al. [2017] which we use for comparison. Figure 14 shows the results of our comparison and they agree with our expectations: the JAM is at least as weak as ARMv8 except in the case of load-buffering (LB).

Many of the results owe to the fact that the JAM is not multi-copy atomic. That is, unlike ARMv8, different threads can see writes to different locations in different orders. So, IRIW+* and WRC+* are allowed for the JAM but not for ARMv8. The `WW+RR+WW+RR+wsilp+poaa+wsilp+poaa` litmus test is a variant of IRIW where all writes are release writes and all reads are aquire reads. The writes in this test use a weaker form of synchronization than the example in Figure 12a in Section 4.4 where the behavior is allowed by the JAM. As a result this behavior is allowed by the JAM. Finally, the LB test is identical to the example in Figure 3c from Section 3. This execution is allowed by the ARMv8 model but it is a cycle in `(po | rf) & opq` which is explicitly disallowed by the JAM.

5.2 Comparison with RC11

C11's atomic memory accesses can be annotated with memory orders. Herd provides the following built-in relations for the memory orders and accesses in the C11 litmus tests: `M` for plain memory access, `RLX` for relaxed memory order accesses, `REL` for release memory order accesses, `ACQ` for acquire memory order accesses, `REL_ACQ` for release-acquire memory order read-write accesses, `SC` for sequentially consistent memory order accesses, `F & REL` for release fences, `F & AQR` for acquire fences, `F & SC` for sequentially consistent (full) fences, and `F \ SC` for all other fences.

```
let opq = RLX | ACQ | REL | ACQ_REL | SC
let rel = REL | ACQ_REL | SC
let acq = ACQ | ACQ_REL | SC
let vol = SC
let svo = po;[F & REL];po;[W] | [R];po;[F & ACQ];po
let spush = po;[F & SC];po
```

Mapping. Our mapping for the relations provided by Herd for C11 follows the informal relationship outlined in the documentation for the JAM [Lea 2018]. All plain accesses in C11 are treated as plain accesses in our mapping to the JAM. We map relaxed memory order accesses or stronger to opaque mode, `opq = RLX | ...` , release memory order accesses or stronger to release mode, `rel = REL | ...` , acquire memory order accesses or stronger to acquire mode, `acq = ACQ | ...` . sequentially consistent memory order accesses to volatile mode, `vol = SC`. We also map release fences before writes, `po;[F & REL];po;[W]`, and acquire fences after reads, `[R];po;[F & ACQ];po`, to specified visibility orders, `svo`. Finally, we map sequentially consistent fences, `po;[F & SC];po`, to push orders between program order earlier and program order later accesses.

Expectations. The access modes of the JAM are inspired by the memory orders of C11 so this comparison is of particular importance. We expect the JAM to match C11 except in cases where release-sequences, consume-reads, or causal cycles are involved.

Results. In Figure 16 we have the results of the comparison with the RC11 model of [Lahav et al. 2017]. We include the tests from research by Wickerson and Batty [2015], [Vafeiadis et al. 2015], [Lahav et al. 2017]. In the case of [Wickerson and Batty 2015] and [Vafeiadis et al. 2015] we used the tests directly. In the case of [Lahav et al. 2017] we translated the tests from the paper to Herd litmus tests ourselves. The results largely agree with our expectations. The exceptions are places where the RC11 model breaks with the C11 specification. We will discuss each in turn.

The `cyc_na` test is the same as the load-buffering example from Section 3 but all the accesses are plain. The JAM allows this cycle in plain mode `po | rf` because its acyclicity requirement only applies to opaque mode or stronger accesses. The RC11 model breaks with C11 by including an acyclicity requirement for `po | rf` for all memory accesses.

The `lb` test is the same as `cyc_na` except that all of the accesses are relaxed memory order. In the mapping to the JAM this translates into opaque mode accesses which are not allowed to exhibit a cycle in `po | rf`. Thus both models forbid the load-buffering behavior in this case.

The `mp_relacq_rs` test leverages the release sequences of C11. Since the JAM does not include release sequences it does not forbid the behavior of this test.

Vafeiadis et al. [2015]

name	RC11	JAM
lb	**Never**	**Never**
fig6	Never	**timed out**
fig6_translated	Never	**timed out**

Lahav et al. [2017]

name	RC11	JAM
IRIW-acq-sc	**Sometimes**	**Never**
Z6.U	**Sometimes**	**Never**

New C11 Tests

name	RC11	JAM
IRIW-sc-rlx-acq	**Sometimes**	**Never**

Wickerson and Batty [2015]

name	RC11	JAM
mp_relacq_rs	**Never**	**Sometimes**

Wickerson and Batty [2015]

name	RC11	JAM
cppmem_iriw_relacq	Sometimes	Sometimes
cppmem_sc_atomics	Never	Never
iriw_sc	Never	Never
mp_fences	Never	Never
mp_relacq	Never	Never
mp_relacq_rs	**Never**	**Sometimes**
mp_relaxed	Sometimes	Sometimes
mp_sc	Never	Never

Lahav et al. [2017]

name	RC11	JAM
2+2W	Never	Never
IRIW-acq-sc	**Sometimes**	**Never**
RWC+syncs	Never	Never
W+RWC	Never	Never
Z6.U	**Sometimes**	**Never**

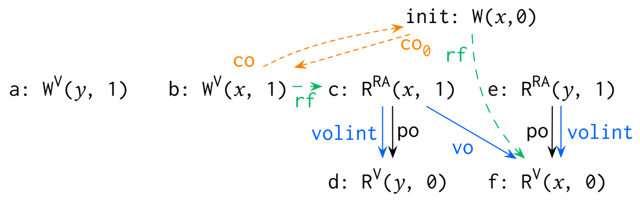
Herd X86 Tests

name	x86	JAM
CoRWR	Never	Never
SB+mfences	Never	Never
4.SB	Sometimes	Sometimes
iriw-internal	Sometimes	Sometimes
podrw000	Sometimes	Sometimes
podrw001	Sometimes	Sometimes
SB	Sometimes	Sometimes
SB+rfi-pos	Sometimes	Sometimes
SB+SC	Sometimes	Sometimes
X000	Sometimes	Sometimes
X001	Sometimes	Sometimes
X002	Sometimes	Sometimes
X003	Sometimes	Sometimes
X004	Sometimes	Sometimes
X005	Sometimes	Sometimes
X006	Sometimes	Sometimes
6.SB	Sometimes	**timed out**
6.SB+prefetch	Sometimes	**timed out**
iriw	Never	Sometimes
x86-2+2W	Never	Sometimes

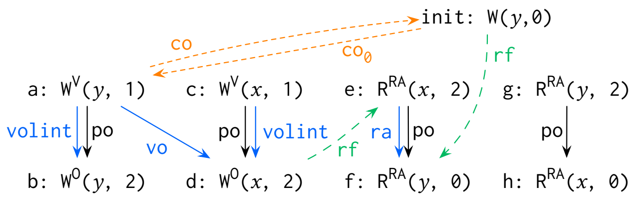
Fig. 16. RC11 & x86 Litmus Test Comparison

Our test runs for `fig6` and `fig6_translated` timed out at 5 minutes. The behavior modeled by these tests highlights a quirk in C11's rules for SC accesses. Together they demonstrate that strengthening the memory order of a particular relaxed store in `fig6` to an SC store in `fig6_translated` creates new behaviors. That is, the tests demonstrate that the memory orders of C11 are not monotonic. RC11 includes a fix proposed by [Vafeiadis et al. 2015] which forbids this behavior. In Section 6 we prove that the JAM's access modes are indeed monotonic in our model which means that stronger access modes exhibit fewer behaviors, thus the behavior in these tests is forbidden.

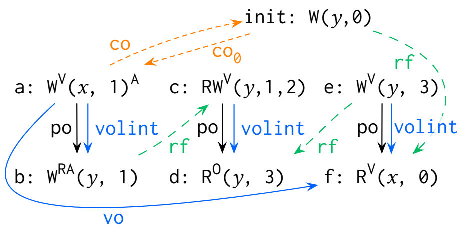
In the case of `IRIW-acq-sc`, `Z6.U`, and `IRIW-sc-rlx-acq` our model forbids the behavior in keeping with the C11 specification. We will consider each case and discuss why RC11 does not forbid each behavior.



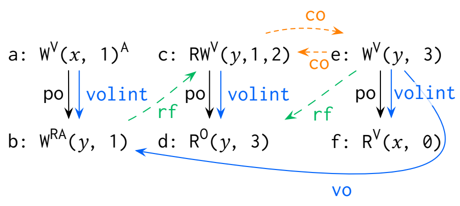
(a) IRIW RA and Volatile Reads



(b) IRIW Extra Writes



(c) Z6.U a $\xrightarrow{\texttt{vo}}$ f



(d) Z6.U e $\xrightarrow{\texttt{vo}}$ b

Fig. 17. Z6.U & IRIW RA with Volatile Reads Litmus Tests

The `IRIW-acq-sc` test appears in in Figure 17a. The reason this is forbidden in our model is that all accesses before a SC/Volatile read are ordered by `volint`. Then, because there are two such orders in the reading threads there is either a visibility order, c $\xrightarrow{\texttt{vo}}$ f or d $\xrightarrow{\texttt{vo}}$ e. Thus, one of the two reads must see the non-initialization write using the same reasoning from Section 4.4 for Figure 12b. This effectively emulates the "leading" fence compilation scheme described in [Lahav et al. 2017], where a full fence is placed before the SC/Volatile accesses. The authors (personal communication) point out that this scheme should forbid this behavior. By contrast RC11 relaxes the C11 model to accommodate a leading *or* trailing fence compilation scheme. In this case, if the trailing fence scheme is used there's no ordering provided to any of the SC accesses and the execution is allowed.

Two possible executions for `Z6.U` test appears in Figures 17c and 17d. They represent each case of the visibility orders induced by a $\xrightarrow{\texttt{volint}}$ b and e $\xrightarrow{\texttt{volint}}$ f. In 17c we have a $\xrightarrow{\texttt{vo}}$ f, which means that f must have seen a and could not read from the initialization by `cowr`. In 17d we have e $\xrightarrow{\texttt{vo}}$ b. We also have b $\xrightarrow{\texttt{rf}}$ c. Then together we have e $\xrightarrow{\texttt{vo}}$ b $\xrightarrow{\texttt{rf}}$ c. Then because c and e are both writes to y we have e $\xrightarrow{\texttt{co}}$ c by `coww`. Separately, since c $\xrightarrow{\texttt{volint}}$ d we have c $\xrightarrow{\texttt{vo}}$ d. Then since e $\xrightarrow{\texttt{rf}}$ d we have c $\xrightarrow{\texttt{co}}$ e by `cowr`. Thus we have a cycle in `co` and the execution if forbidden.

This effectively emulates the "trailing" fence compilation scheme described by Lahav et al. [2017], where a full fence is placed after SC/Volatile accesses. They point out that this scheme should forbid this behavior. Again, RC11 relaxes C11 to model both schemes and under a leading fence compilation scheme the behavior this allowed.

Finally, in Figure 17b we have our own variant of IRIW called `IRIW-sc-rlx-acq` based on the `WW+RR+...` test in the ARMv8 suite. In this case, if the compiler is inserting trailing fences after a and c, then either a $\xrightarrow{\texttt{vo}}$ d or b $\xrightarrow{\texttt{vo}}$ c. Taking the first case we can can construct b $\xrightarrow{\texttt{vo+}}$ f from

$a \xrightarrow{vo} d \xrightarrow{rf} a \xrightarrow{ra} f$ and a cycle in co by cowr. The second case is similar but with the read of x in the 4th thread.

Again, the RC11 model does not forbid this execution. In our discussion with the authors, they suggested two possible interpretations for this behavior. The first is a leading fence compilation scheme which we have discussed above. The second is a merge of each pair of writes into a single write. In the second case we suggest that the Java compiler should not merge opaque mode (or stronger) accesses.

In summary, our model correctly forbids the behavior in each of these cases. The reason is that volatile reads would be preceded by a full fence and volatile writes would be followed by a full fence in the presence of mixed mode programs with SC accesses. Importantly, the tension between optimal compilation and a simple model exists here as it does with C11.

5.3 Comparison with x86

Herd provides the following built in relations for x86 litmus tests: `M` for memory accesses, `SFENCE` for fences for intra-thread ordering of writes with other accesses, `LFENCE` for ordering intra-thread ordering of reads with other accesses, and `MFENCE` as a full fence with cross thread ordering guarantees.

```
let opq = M
let svo = [W];po;[SFENCE];po | [R];po;[LFENCE];po
let spush = po;[MFENCE];po
```

Mapping. We map from all regular memory accesses to opaque mode `opq = M` in keeping with the opaque mode mapping from our comparison with ARMv8. We map `SFENCE` to specified visibility orders from writes to other memory accesses across the fence, `[W];po;[SFENCE];po`. We map `LFENCE` to specified visibility orders from reads to other memory accesses across the fence, `[R];po;[LFENCE];po`. Finally we map `MFENCE` as a specified push orders between any access before the fence to any access after the fence, `po;[MFENCE];po`.

Expectations. We expect the JAM to be weaker than x86 in all cases. The only weak behavior that x86 exhibits is reordering writes with reads (store-buffering) which the JAM also allows in opaque mode.

Results. Figure 16 shows the results of our comparison with the x86 model included with Herd. The litmus tests are also included with Herd for the model. The two tests which timed out, `6.SB+prefetch` and `6.SB`, are store buffering variants. Given that the JAM can reorder writes with reads the behavior of these tests is allowed. Otherwise the tests confirm our expectations.

6 METATHEORY

Here, we develop a metatheory for our model of the JAM. First, we detail the semantics and then sketch the proofs for the theorems listed in the introduction. We show that each mode, from Volatile to Opaque, admits strictly more executions, otherwise referred to as monotonicity. We also show that properly synchronized programs, i.e. race-free programs as defined by Boehm and Adve [2008], will only exhibit sequentially consistent behaviors as required by Java's data-race-free (DRF) guarantee. Finally, we show that acquire mode for all reads obviates the inclusion of the JAM's acyclic causality requirement.

These theorems further validate the definitions of our model, give more evidence that the model is complete with respect to the documentation, and clearly demonstrate that the semantics is suitable for formal reasoning. The semantics, lemmas and theorems have been mechanized in Coq. The source is available for inspection in the supplementary material and it includes instructions on where to find everything included below.

6.1 Semantics

The axioms of our model apply to memory event graphs, each of which represents one possible execution of a program. Thus far we have relied on Herd to generate executions for a program and then apply the axioms of our model to rule on whether an execution is allowed. Our formalization in Coq replaces the Herd machinery with the history fragment of the semantics of Crary and Sullivan [2015] to model the set of possible executions. Here we will give enough detail about the history semantics to understand the content of our theorems and proofs. We provide a more comprehensive set of definitions and intuition for the semantics in Appendex C. The full semantics is available in the supplementary material.

We use n and l to range over natural numbers and memory locations. We use i to range over unique memory access identifiers where previously we have used a, b, c, etc. We use m to represent one of the four access modes, P for plain, O for opaque, RA for release acquire and V for volatile. Memory accesses, a, can take the form of reads, l_m, and writes, $l_m := n$ with their accompanying modes as well as read-writes, $\mathsf{RW}(l, n)$.

We use H to range over lists of memory events that represent executions. We call these lists histories and we use $H(h)$ means that the memory event h is in H. Memory events record the program's interactions with a history. For example, $H(\mathsf{is}(i, l_m := n))$ records that the identifier i is a write to l of the value n and $H(\mathsf{exec}(i))$ records that i has executed, subject to the axioms of the memory model. We use $i \xrightarrow{\mathsf{R}}_H i$ to represent our model's relations as they apply to H.

We also require that histories satisfy some basic well-formedness conditions. For example, a read must take it's value from a write to the same location which has been executed. Importantly, we require that the trace order respects any intra-thread ordering created by visibility generated from specified orders, release-acquire accesses, or volatile accesses. This is in keeping with our restriction on trace order as detailed in Section 4.2. We call this well-formedness condition executable. In what follows, the well-formedness conditions are unified as *trace coherence*. Otherwise, the only restrictions on histories come from the acyclicity conditions outlined in Section 4.1.

6.2 Theorems

To begin, we demonstrate the monotonicity of our access mode definitions. We define the reflexive ordering of the access modes as $\mathsf{P} \sqsubseteq \mathsf{O} \sqsubseteq \mathsf{RA} \sqsubseteq \mathsf{V}$ and extend it to accesses $l_{m_1} \sqsubseteq l_{m_2}$, $l_{m_1} := n_1 \sqsubseteq l_{m_2} := n_2$, $\mathsf{RW}(l, n_1) \sqsubseteq \mathsf{RW}(l, n_2)$ whenever $m_1 \sqsubseteq m_2$. As a technical matter we treat read-writes as always having the same order. We extend the order to histories by matching identifiers and ordering the accesses.

$$H_1 \sqsubseteq H_2 \triangleq \forall\, i\; a_1\; a_2, H_1(\mathsf{is}(i, a_1)) \wedge H_2(\mathsf{is}(i, a_2)) \Rightarrow a_1 \sqsubseteq a_2$$

When the po, rf, and to relations of two histories H_1 and H_2 have the following relationships: $\xrightarrow{\mathsf{po}}_{H_2} \subseteq \xrightarrow{\mathsf{po}}_{H_1}$, $\xrightarrow{\mathsf{to}}_{H_2} \subseteq \xrightarrow{\mathsf{to}}_{H_1}$, $\xrightarrow{\mathsf{rf}}_{H_2} \subseteq \xrightarrow{\mathsf{rf}}_{H_1}$, then we say they *match*.

THEOREM 1 (MONOTONICITY). *For two histories H_1 and H_2, suppose that both match, both are trace coherent, and $H_2 \sqsubseteq H_1$. Further suppose that* $\mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H_1})$ *and that there are no specified visibility orders or push orders in H_2, then* $\mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H_2})$

We make two notes. First the absence of specified orders in H_2 is a technical convenience since specified order edges are not related to the strength of the access modes for reads and writes. Second, we focus on the acyclic coherence requirement because the match assumption means that it would be trivial to satisfy the acyclic causality requirement for H_2 supposing it is true of H_1 because po and rf have fewer edges in H_2.

Proof Sketch. We assume $i \xrightarrow{\text{co}}_{H_2} i$ for some i and show that this must mean $i \xrightarrow{\text{co}}_{H_1} i$ which is a contradiction. This is straight forward by induction on $i \xrightarrow{\text{co}}_{H_2} i$, noting that each case of visibility in H_2 will exist in H_1 because of stronger access modes in H_1. □

For the last two theorems will first establish that the main component of visibility in our model, vvo, is irreflexive.

Lemma 1 (Irreflexive Visibility). *If H is trace coherent then, for all i, $\neg\, i \xrightarrow{\text{vvo+}}_{H} i$.*

Proof Sketch. This follows from two facts. First, vvo derives from orderings that are always either program ordered from intra-thread synchronization (svo, spush, ra, volint) or trace ordered (pushto, rf). Second, both relations are total and to is consistent with the po edges for intra-thread visibility by the executable well-formedness condition. □

Next we show that if a program is properly synchronized then it will only exhibit sequentially consistent behavior. We require the following standard definitions including the traditional notion of sequential consistency [Shasha and Snir 1988]:

$$i_1 \xrightarrow{\text{fr}}_H i_2 \triangleq \exists\, i_3, i_3 \xrightarrow{\text{rf}}_H i_1 \wedge i_3 \xrightarrow{\text{co}}_H i_2$$
$$i_1 \xrightarrow{\text{com}}_H i_2 \triangleq i_1 \xrightarrow{\text{co}}_H i_2 \vee i_1 \xrightarrow{\text{rf}}_H i_2 \vee i_1 \xrightarrow{\text{fr}}_H i_2$$
$$i_1 \xrightarrow{\text{sc}}_H i_2 \triangleq i_1 \xrightarrow{\text{po}}_H i_2 \vee i_1 \xrightarrow{\text{com}}_H i_2$$

We also require a definition of proper synchronization in keeping with our focus on visibility.

$$i_1 \xrightarrow{\text{sync}}_H i_2 \triangleq \exists i_3, i_1 \xrightarrow{\text{vvo+}}_H i_3 \xrightarrow{\text{po*}}_H i_2 \wedge \forall i_4, i_3 \xrightarrow{\text{po*}}_H i_4 \Rightarrow i_3 \xrightarrow{\text{vvo}}_H i_4$$

The idea is that any conflicting access is visibility ordered by some mechanism, be it a specified order (fence) or a strong mode for i_3. Then, following the definition of "type 2" data-races from Boehm and Adve [2008], we say that H is *race-free* when, for all conflicting accesses i_1 and i_2, we have $i_1 \xrightarrow{\text{sync}}_H i_2$ or $i_2 \xrightarrow{\text{sync}}_H i_1$.

Theorem 2 (DRF-SC). *If H is trace coherent, race free and,* acyclic($\xrightarrow{\text{co}}_H$)*, then* acyclic($\xrightarrow{\text{sc}}_H$).

Proof Sketch. We assume $i \xrightarrow{\text{sc+}}_H i$ for some i and show a contradiction. By Lemma 1 it is enough to demonstrate a cycle in vvo+.

We can show that for any i_1 and i_2, if we have $i_1 \xrightarrow{\text{com}}_H i_2$ then we have $i_1 \xrightarrow{\text{sync}}_H i_2$. Note that any accesses related by com are conflicting. Then we have either $i_1 \xrightarrow{\text{sync}}_H i_2$ or $i_2 \xrightarrow{\text{sync}}_H i_1$. In each case for com we can show that $i_2 \xrightarrow{\text{sync}}_H i_1$ creates a cycle in the coherence order so it must be $i_1 \xrightarrow{\text{sync}}_H i_2$.

Then, since po is irreflexive, we have that any sequence $i \xrightarrow{\text{sc+}}_H i$ must include at least one com edge. Then since com edges are also sync edges, when we have $i_1 \xrightarrow{\text{sc+}}_H i_2$ we also have $i_1 \xrightarrow{\text{po}\,|\,\text{com+}}_H i_2$ and therefore $i_1 \xrightarrow{\text{po}\,|\,\text{sync+}}_H i_2$.

But then we can rearrange a cycle in $i \xrightarrow{\text{po}\,|\,\text{sync+}}_H i$ to be $i \xrightarrow{\text{sync+}}_H i$ by appending a leading po edge to the end. Observe that for any sequence $i_1 \xrightarrow{\text{sync+}}_H i_2$ we have $i_1 \xrightarrow{\text{vvo+}}_H i_2$, so we have $i \xrightarrow{\text{vvo+}}_H i$ as required. □

Finally, we show that if all reads are acquire-reads then the JAM's acyclic causality requirement is unnecessary. This theorem demonstrates the soundness of proposed compiler implementations for satisfying the acyclic causality requirement of the JAM [Ou and Demsky 2018].

Theorem 3 (Causal Acquire-Reads). *If H is trace coherent and all reads in H are acquire-reads, then* acyclic($\xrightarrow{\text{po}\,|\,\text{rf}}_H$).

Proof Sketch. We assume $i \xrightarrow{\text{po} \mid \text{rf}+}_H i$ for some i and show a contradiction. By Lemma 1, it is enough to demonstrate a cycle in vvo+.

First, note that any sequence $i_1 \xrightarrow{\text{rf}}_H i_2 \xrightarrow{\text{po}}_H i_3$ implies $i_1 \xrightarrow{\text{vvo}+}_H i_3$ because rf implies vvo and because, by assumption, the read i_2 is an acquire read and i_3 is program order later, so $i_2 \xrightarrow{\text{vvo}}_H i_3$.

Let $\xrightarrow{\text{rfpoq}}_H$ be a reads-from edge followed by an optional program order edge. Note that, by induction and the fact that rf and rfpo imply vvo we can show that $i_1 \xrightarrow{\text{rfpoq}}_H i_2$ implies $i_1 \xrightarrow{\text{vvo}+}_H i_2$

Then, since po is irreflexive, we have that any sequence $i \xrightarrow{\text{po} \mid \text{rf}+}_H i$ must include at least one rf edge. Thus we can rearrange to get $i' \xrightarrow{\text{rfpoq}+}_H i'$ and we have $i' \xrightarrow{\text{vvo}+}_H i'$ by the above, as required. □

7 UNOBSERVABLE TOTAL COHERENCE ORDER

Here we will demonstrate that the effects of a total coherence order are unobservable in our model. This is a surprising result since a total coherence order is intuitively associated with maintaining per-location state and we would expect concurrent writes to have a material impact on the behavior of programs.

To start, recall that a feature of a memory model as exhibited by an example program is defined in binary terms. The feature is present when any executions are allowed and the feature is absent when no executions are allowed. Then we say that a feature is *observable* when we can construct any example program such that the feature's presence in the model changes whether any executions are allowed. So, to observe a total coherence order, we must construct a program that changes whether executions are allowed based on the presence of the total order. We will show that this is impossible under our model.

First, note that in our model there is no way to allow previously forbidden executions when adding coherence order edges. This means we can't observe the feature by going from some executions *with* a total order to no executions *without* a total order. Thus, any program that allows us to observe the total coherence order must forbid all executions in the presence of a total order. We will show that such a program will also forbid all executions when the total order is removed.

Theorem 4. *It is impossible to construct an example program for the JAM, such that, for any two writes, if they are totally ordered there are no valid executions, and if they are not totally ordered there is at least one valid execution.*

Proof. From a total order we have that all candidate executions can be divided between the two directions. For some writes to the same location a and b we have:

$$\mathsf{a} \xrightarrow{\mathsf{co}} \mathsf{b} \lor \tag{1}$$

$$\mathsf{b} \xrightarrow{\mathsf{co}} \mathsf{a} \tag{2}$$

We begin by eliminating all executions for the first direction, (1). This means we must construct a cycle in co, such that, for all executions including (1) we have :

$$\mathsf{a} \xrightarrow{\mathsf{co}} \mathsf{b} \tag{1}$$

$$\mathsf{b} \xrightarrow{\mathsf{co}} \cdots \xrightarrow{\mathsf{co}} \mathsf{a} \tag{3}$$

With this cycle, executions including (1) are forbidden by the acyclicity requirement for co. Observe that (3) must be present in all executions of the program. If it is not then we can divide the executions that do not have (3) between executions with (1) and executions with (2). For the executions without (3) and with (1) the execution is allowed and we have failed to forbid all executions. Now we must eliminate all executions for (2) with another cycle:

$$\mathsf{b} \xrightarrow{\mathsf{co}} \mathsf{a} \tag{2}$$

$$\mathsf{a} \xrightarrow{\mathsf{co}} \cdots \xrightarrow{\mathsf{co}} \mathsf{b} \tag{4}$$

Moreover (4) must persist for all executions for the same reason that (3) persisted. Then it must be that, for all executions we have:

$$b \xrightarrow{co} \cdots \xrightarrow{co} a \tag{3}$$

$$a \xrightarrow{co} \cdots \xrightarrow{co} b \tag{4}$$

Then, even if we remove the total order, (1) and (2), from the model, all executions of the program will still have a forbidden cycle by (3) and (4) and we can not demonstrate a single execution that is allowed. □

Critically, this line of reasoning depends on the fact that we have a single acyclicity requirement in which co participates and both (3) and (4) form a cycle that violates that requirement. If that were not the case then (3) and (4) would not necessarily create a cycle or forbid any executions. As a result this reasoning does not apply to the ARMv8 and RC11 models as they have multiple acyclicity requirements involving the coherence order. In Appendix D we have included a litmus test for ARMv8 that shows it is possible to construct a program for that model that behaves differently with and without the total order.

Demonstrating that a total coherence order is not observable in our model means that the JAM may adopt a total coherence order without affecting the outcomes of the model. This would allow it to emulate other mainstream memory models in this respect and recover the intuitive notion of per-location state.

8 RELATED WORK

The JAM is intended as an update and expansion of Java's memory model. As a result, we have taken a fresh look at how to construct our model, but our work takes inspiration from a large body of research on such formalization efforts.

8.1 The Original Java Memory Model

The original Java Memory Model [Manson et al. 2005] included an early attempt to model standard compiler optimizations while ruling out thin-air reads. The specification in that work was part prose and part formal definitions and the "causality" mechanisms at the heart of the model made the definitions complex. Taken together these issues made the model unsuitable for the tasks which one normally formalizes a programming language, namely accurate discourse, metatheory, and algorithm verification. Later work by Ševčík and Aspinall [2008] manually examined a large suite of litmus tests to show that the model disallowed some standard compiler optimizations. Eventually the model was fully formalized in Coq by Huisman and Petri [2007] and Aspinall and Ševčík [2007], but, to the best of our knowledge, there is no way to easily test the behavior of example programs. By contrast we have constructed a readable and testable model for Java's new access modes.

8.2 C11, ARM, and x86

The C11 memory model, which served as the inspiration for Java'a access modes, has seen extensive study. It was originally formalized by Batty et al. [2011] with later revisions to include read-modify-writes and fences [Sarkar et al. 2012]. The work of Vafeiadis et al. [2015], from which we draw the largest set of our C11 litmus tests, studied the soundness of common compiler optimizations under the model of C11 by Morisset et al. [2013]. Our monotonicity theorem is modeled after the same theorem from Vafeiadis et al. [2015]. Most recently, the axiomatic model of Lahav et al. [2017] incorporated the proposed fixes of Vafeiadis et al. [2015] and addressed the unsound compilation strategies which we discussed in Section 5. Also, further progress has been made on new models for C11 that more successfully support standard compiler optimizations without the problem of thin-air reads [Kang et al. 2017; Pichon-Pharabod and Sewell 2016]. However, like the original Java memory model, these models rely on complex formal constructs (promises and event structures respectively). Again, our semantics remains relatively simple thanks to the JAM's broad definition of causal cycles.

As outlined previously there are several important differences between C11 and Java. First, the partial coherence order required careful consideration. The consequences of this design choice manifests most clearly where atomic read-writes are concerened. Second, the lack of legacy features, like C11's release sequences, and the simple mechanism that forbids causal cycles allowed us to build a simple model. In turn, the simplicity of the model makes it more readable than existing C11 models and it allowed us to argue forcefully that the model should adopt a total coherence order.

These differences appear most clearly in our litmus test comparison with RC11 model of Lahav et al. [2017]. Of particular note are the `mp_relacq_rs` and `lb` tests. In the first case the JAM is weaker than C11 because it does not support release sequences, in the second case it is stronger because the C11 specification gives no concrete definition for how to rule out thin-air reads. Notably, the RC11 herd model also forbids causal cycles in `po | rf` so the load buffering behavior is forbidden. This method of preventing thin-air reads in RC11 is included to enable their proofs of soundness for compilation to POWER and not as a representation of the C11 specification.

Programs that mix SC/volatile access modes are the primary focus of Lahav et al. [2017]. In particular the leading and trailing fence insertion approach to compilation was shown to be unsound for Power under models prior to RC11. We compared our model with RC11 in Section 5. Our semantics correctly forbids the behavior described in `IRIW-sc-rlx-acq`, `IRIW-aqc-sc` and `Z6.U` in keeping with the JAM documentation.

Hardware memory models have also seen extensive study. x86 was studied by Owens et al. [2009] and Alglave et al. [2014]. We use the model included with Herd in our litmus test comparisons and we saw that x86 was stronger than the JAM, as expected. ARM processors have traditionally had a much weaker memory model when compared with x86. Recently the model for ARMv8 [Pulte et al. 2017] expanded the guarantees made by the architecture to include multi-copy-atomicity. We saw the effects of this in the behavior of the `IRIW-*` and `WRC-*` litmus tests from our comparison between the JAM and ARMv8. As expected the JAM is weaker than ARMv8 in every case except where cycles in `po | rf` are concerened.

8.3 The Relaxed Memory Calculus

Our mechanized semantics is based on the history fragment of the Relaxed Memory Calculus of Crary and Sullivan [2015]. We use their concept of specified push orders and we draw inspiration for our definitions from their notion of visibility. Also, the proof of our theorems benefited greatly from the library of lemmas included with the RMC mechanization. However, their purpose was to model a weaker version of C11 in the interest of generality while, our goal is to model the JAM. Importantly, we do not employ the execution orders of RMC, our coherence definition is far more compact and we have added the `corr` rule to follow a more standard notion of coherence.

9 CONCLUSION

We have presented the first formal model for Java's access modes. Our model is precise, complete, and testable. We have validated our model against the expected behavior of example programs relative to three other mainstream memory models. We have further validated the model by using it to prove general theorems about the semantics of the Java access modes. Finally, we used our model to demonstrate that Java's access modes can adopt a total coherence order.

We believe our model is a strong foundation for a larger formalization effort around weak-memory concurrency on the Java platform. In particular we would like expand the model and use it to verify the lock-free algorithms of the standard library.

REFERENCES

Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2008. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming (DAMP '09)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1481839.1481842

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

David Aspinall and Jaroslav Ševčík. 2007. Formalising Java's Data Race Free Guarantee. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07)*. Springer-Verlag, Berlin, Heidelberg, 22–37. http://dl.acm.org/citation.cfm?id=1792233.1792237

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. *SIGPLAN Not.* 46, 1 (Jan. 2011), 55–66. https://doi.org/10.1145/1925844.1926394

Mark Batty and Peter Sewell. 2014. The Thin-air Problem. http://www.cl.cam.ac.uk/~pes20/cpp/notes42.html. [Online, accessed Dec 2018].

John Bender, Mohsen Lesani, and Jens Palsberg. 2015. Declarative Fence Insertion. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 367–385. https://doi.org/10.1145/2814270.2814318

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

C++ Standards Committee, Pete Becker, et al. 2010. Programming Languages-C++(final committee draft). C++ standards committee paper WG21/N3092= J16/10-0082.

Jon Corbet. 2012. ACCESS_ONCE(). https://lwn.net/Articles/508991/. [Online, accessed Dec 2018].

Karl Crary and Michael Sullivan. 2015. A Calculus for Relaxed Memory. In *Proceedings of POPL'15, ACM Symposium on Principles of Programming Languages.*

Marieke Huisman and Gustavo Petri. 2007. The Java memory model: a formal explanation. *VAMP* 7 (2007), 81–96.

JDK9. 2017. VarHandle (Java SE 9 & JDK 9). https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html. [Online, accessed March 2019].

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of POPL'17, SIGPLAN–SIGACT Symposium on Principles of Programming Languages.*

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

Doug Lea. 2017. JEP 193. http://openjdk.java.net/jeps/193. [Online, accessed March 2019].

Doug Lea. 2018. Using JDK 9 Memory Order Modes. http://gee.cs.oswego.edu/dl/html/j9mm.html. [Online, accessed March 2019].

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++ 11 memory model. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 187–196.

Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-thin-air Results. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 136 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276506

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: X86-TSO. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. *SIGPLAN Not.* 51, 1 (Jan. 2016), 622–633. https://doi.org/10.1145/2914770.2837616

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158107

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising c/c++ and power. *Acm Sigplan Notices* 47, 6 (2012), 311–322.

Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–51. https://doi.org/10.1007/978-3-540-70592-5_3

Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 282–312. https://doi.org/10.1145/42190.42277

Michael J. Sullivan. 2015. Low-level Concurrent Programming Using the Relaxed Memory Calculus. https://www.msully.net/stuff/thesprop.pdf. PhD. Dissertation [Online, accessed July 2017].

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

John Wickerson and Mark Batty. 2015. Taming the complexities of the C11 and OpenCL memory models. *arXiv preprint arXiv:1503.07073* (2015).

A FULL HERD MODEL

```
let opq = O | RA | V
let rel = W & RA
let acq = R & RA
let vol = V

(* release acquire ordering *)
let ra = po;[rel] | [acq];po

(* intra-thread volatile ordering *)
let volint =  po;[vol & R] | [vol & W];po

(* intra-thread ordering contraints *)
let into = svo | spush | ra | volint

(* define trace order, ensure it respects rf and intra-thread specified orders *)
(* Note that ((W * FW) & loc & ~id) = cofw *)
with to from linearisations(M\IW, ((W * FW) & loc & ~id) | rf | into)

(* cross thread push ordering extended with volatile memory accesses *)
let push = spush | volint
let pushto = to+ & (domain(push) * domain(push))

(* extend ra visibility *)
let vvo = rf | svo | ra | push | pushto;push
let vo = vvo+ | po-loc

include "filters.cat"
let WWco(rel) = WW(rel) & loc & ~id
let cofw = WWco((W * FW))

(* coherence rules *)
let coinit = loc & IW*(W\IW)
let coww = WWco(vo)
let cowr = WWco(vo;invrf)
let corw = WWco(vo;po)
let corr = WWco(rf;po;invrf)

(* general definition from RC11, works for atomic rws and split instruction rws *)
let rmw-jom = [RMW] | rmw

(* read-write rules *)
let cormwtotal = WWco(((range(rmw-jom) * _ ) | (_ * range(rmw-jom))) & to)
let cormwexcl = WWco((rf;rmw-jom)^-1;co-jom)

let rec co-jom = coww | cowr | corw  | corr
               | cofw | coinit | cormwtotal
               | WWco((rf;rmw-jom)^-1;co-jom)

acyclic (po | rf) & opq
acyclic co-jom
```

B SPECIFIED ORDERS: A MAPPING FOR ARMV8 AND X86 READ-WRITES

Specified orders can be compiled to existing synchronization using a fairly direct mapping. Here we give an example mapping for ARMv8. We also discuss how specified visibility orders can be elided when the target platform for compilation already enforces them.

$$[\mathsf{W}];\mathsf{svo};[\mathsf{M}] \rightsquigarrow [\mathsf{W}];\mathsf{po};[\mathsf{DMB\ ST}];\mathsf{po};[\mathsf{M}]$$
$$[\mathsf{R}];\mathsf{svo};[\mathsf{M}] \rightsquigarrow [\mathsf{R}];\mathsf{po};[\mathsf{DMB\ LD}];\mathsf{po};[\mathsf{M}]$$
$$[\mathsf{M}_1];\mathsf{push};[\mathsf{M}_2] \rightsquigarrow [\mathsf{M}_1];\mathsf{po};[\mathsf{DSB}];\mathsf{po};[\mathsf{M}_2]$$

Note that this mapping is informal. The fence instruction for each mapping must be inserted on *all* program order paths between the ordered instruction as demonstrated originally by Bender et al. [2015] and Sullivan [2015].

The value of these orders can be seen in the case of atomic read-writes on x86. The JAM makes no intra-thread ordering guarantees for atomic read-writes even though they exist on some architectures like x86 [Owens et al. 2009]. This might result in unnecessary synchronization to achieve a desired outcome that requires such intra-thread ordering. However, using specified orders means that a compiler can make intelligent decisions based on the target platform. For example if we have a specified visibility order between a read-write and a later read target architecture is x86, the compiler can recognize that the head of the order is a compare and exchange instruction and omit any extra synchronization:

$$[\mathsf{RW}];\mathsf{svo};[\mathsf{R}] \rightsquigarrow [\mathsf{RW}];\mathsf{po};[\mathsf{R}]$$

C HISTORY SEMANTICS

The syntax of our formalization appears in Figure 18. We use n, l, i, and p to range over natural numbers, memory locations, unique memory access identifiers, and unique thread identifiers. We use m to represent one of the four access modes, P for plain, O for opaque, RA for release acquire and V for volatile. Note that RA writes are release writes and RA reads are acquire reads. Memory accesses, a, can take the form of reads, l_m, and writes, $l_m := n$ with their accompanying modes as well as read-writes, $\mathrm{RW}(l, n)$.

Memory events, h record the program's interactions with memory. Notably, we assume that the program and history can record specified visibility and specified push orders, $\mathrm{vo}(i, i)$ and $\mathrm{push}(i, i)$ before the execution of the related identifiers. For example, the program may use the labeling and ordering mechanism of Crary and Sulivan [cite]. We discuss the other events in detail below.

We use d to range over program generated events which are translated by the memory semantics into memory events and we use P to abstract over an expression language that can produce such events. We use H to represent a list of memory events, where $H(h)$ means that $h \in H$.

Finally we use $i \xrightarrow{\mathsf{R}}_H i$ to represent memory model relations for H. In what follows, the restriction on the trace order, $\mathtt{to}$, and the acyclicity requirements form the interface with the relations of our axiomatic model.

Program and history states transition together via the step relation defined in figure 19. The history transition semantics certifies program events $d@p$ of the form $i = a@p$, $i@p$, or i to $n@p$ and turns them into memory events.

$i = a@p$ represents the "initialization" of a memory access a using the unique identifier i in thread p. The history semantics appends $\mathsf{init}(i, p)$ and $\mathsf{is}(i, a)$ to H to record the initialization and the form of the memory access identified by i. The only certification required of an initialization is that the memory access identified by i is not already initialized. Importantly, we assume that the program initializes memory accesses in program order, so the subsequence of inititalization events records program order in H.

$$\begin{array}{rlrl}
\text{Nat} & n := 0 \mid 1 \mid \ldots & \text{Accesses} & a := l_m \mid l_m := n \mid \mathsf{RW}(l, n) \\
\text{Locations} & l := \ldots & \text{Mem. Events} & h := \mathsf{init}(i, p) \mid \mathsf{is}(i, a) \mid \mathsf{exec}(i) \\
\text{Modes} & m := \mathsf{P} \mid \mathsf{O} \mid \mathsf{RA} \mid \mathsf{V} & & \quad \mid \mathsf{rf}(i, i) \mid \mathsf{vo}(i, i) \mid \mathsf{push}(i, i) \\
\text{Access Ids} & i := \ldots & \text{Prog. Events} & d := i = a \mid i \mid i \text{ to } n \\
\text{Thread Ids} & p := \ldots & \text{Program} & P := \ldots \\
 & & \text{History} & H := \epsilon \mid H, h
\end{array}$$

Fig. 18. Syntax

$$\frac{P \xrightarrow{d@p} P' \quad H \xrightarrow{d@p} H'}{(P, H) \rightarrow (P', H')}\ \text{step} \qquad \frac{\mathsf{wf}(H, \mathsf{exec}(i), p) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H, \mathsf{exec}(i)})}{H \xrightarrow{i@p} H, \mathsf{exec}(i)}\ \text{write}$$

$$\frac{\neg H(\mathsf{init}(i, \_))}{H \xrightarrow{i=a@p} H, \mathsf{init}(i, p), \mathsf{is}(i, a)}\ \text{init} \qquad \frac{\begin{array}{r}\mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H, \mathsf{rf}(i_w, i)}) \\ \mathsf{wf}(H, \mathsf{rf}(i_w, i), p, n) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{po} \mid \mathsf{rf}}_{H, \mathsf{rf}(i_w, i)})\end{array}}{H \xrightarrow{i \text{ to } n@p} H, \mathsf{rf}(i_w, i)}\ \text{read}$$

$$\begin{aligned}
\mathsf{reads}(H, i, l) &\triangleq \exists\, m, H(\mathsf{is}(i, l_m)) \lor \exists\, n, H(\mathsf{is}(i, \mathsf{RW}(l, n))) \\
\mathsf{writes}(H, i, l, n) &\triangleq \exists\, m, H(\mathsf{is}(i, l_m := n)) \lor H(\mathsf{is}(i, \mathsf{RW}(l, n))) \\
\mathsf{executed}(H, i) &\triangleq H(\mathsf{exec}(i)) \lor \exists\, i_w, H(\mathsf{rf}(i_w, i)) \\
\mathsf{executable}(H, i) &\triangleq \neg\, \mathsf{executed}(H, i) \land \forall i', i' \xrightarrow{\mathsf{into}}_H i \implies \mathsf{executed}(H, i')
\end{aligned}$$

$$\mathsf{wf}(H, \mathsf{exec}(i), p) \triangleq \begin{cases} H(\mathsf{init}(i, p)) \\ H(\mathsf{is}(i, l_m := \_)) \\ \mathsf{executable}(H, i) \end{cases} \quad \mathsf{wf}(H, \mathsf{rf}(i_w, i), p, n) \triangleq \begin{cases} H(\mathsf{init}(i, p)) \\ \mathsf{reads}(H, i, l) \\ \mathsf{writes}(H, i_w, l, n) \\ \mathsf{executed}(H, i_w) \\ \mathsf{executable}(H, i) \end{cases}$$

Fig. 19. Semantics

$i@p$ represents the execution of a write in thread p which is recorded in history with $\mathsf{exec}(i)$. Writes must be certified by the acyclicity requirement for co and a basic well-formedness condition $\mathsf{wf}(H, \mathsf{exec}(i), p)$. The well-formedness requires that i be initialized $H(\mathsf{init}(i, p))$, that the memory access associated with i be a write, $H(\mathsf{is}(i, l_m := n))$, and that the write be executable $\mathsf{executable}(H, i)$.

The $\mathsf{executable}(H, i)$ constraint requires that i has not already executed and that the sequence of execution, to, respects any intra-thread ordering, into, as in the description of Section 4.2. The program is otherwise free to execute memory accesses out-of-order.

i to $n@p$ represents the execution of a read i, reading the value n in thread p which is recorded in history with $\mathsf{rf}(i_w, i)$. Reads must be certified by the same acyclicity requirement for co and the additional requirement on po | rf. The well-formedness condition for reads requires that i be initialized, that it be a read or a read-write, $\mathsf{reads}(H, i, l)$, and that it be executable. It further requires that the paired write i_w is executed, executed(H, i_w), and that i_w writes the value n to the same location l, writes(H, i, l, n).

D OBSERVABLE TOTAL COHERENCE FOR ARMV8

Here we demonstrate that it is possible to construct a program that is only forbidden due to the total coherence order of the ARMv8 memory model from Pulte et al. [2017]. We start by noticing that Herd model for ARMv8 has two acyclicity requirements that involve co:

```
(* Internal visibility requirement *)
acyclic po-loc | ca | rf as internal

(* External visibility requirement *)
irreflexive ob as external
```

In the first requirement `ca = fr | co`. In the second `ob = obs | ...` where `obs = ... | coe` and coe is coherence restricted to inter-thread relationships. Critically, as illustrated in the proof for our model, they do not together work to form cycles. So we can use one with each side of the total order to demonstrate its observability in the model.

We have constructed the following program (included in the supplementary material) which will exhibit a cycle in the first requirement for one side of the total order and a different, incompatible cycle, for the second requirement:

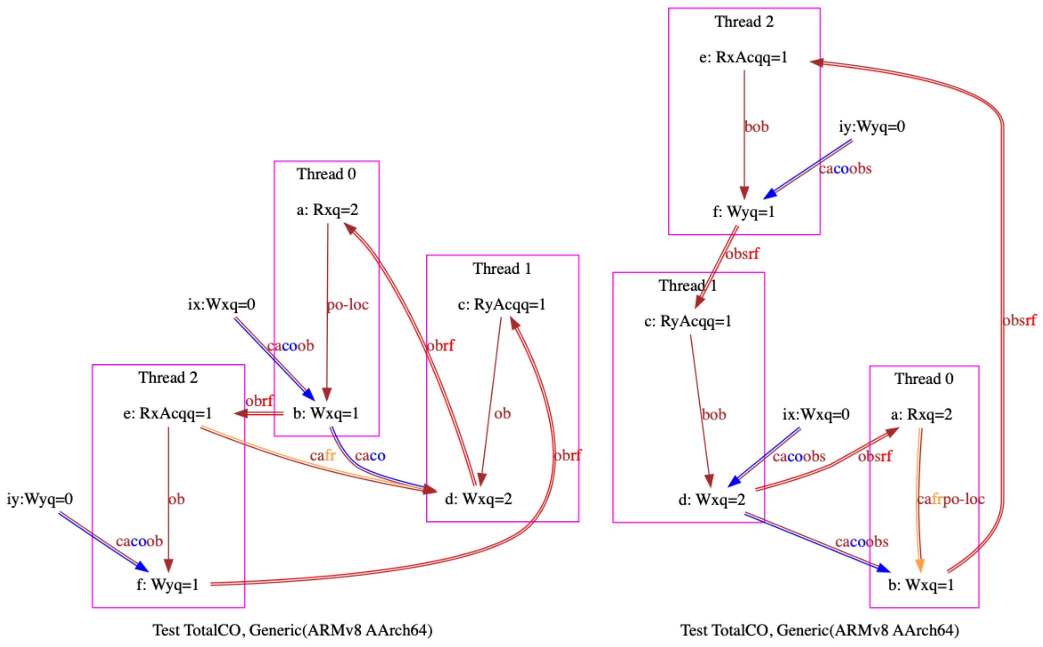
```
Arch64 totalco
{
0:X1=x; 0:X3=y;
1:X1=x; 1:X3=y;
2:X1=x; 2:X3=y;
}
 P0             | P1            | P2;
 LDR X2,[X1]    | LDAR X5, [X3]| LDAR X5,[X1];
 MOV X0,#1      | MOV X2,#2     | MOV X0, #1;
 STR X0,[X1]    | STR X2,[X1]   | STR X0, [X3];

exists (0:X2=2 /\ 1:X5=1 /\ 2:X5=1)
```

We show this by running the program with Herd and allowing executions that violate these two requirements (we mark them with "flag" in the Herd parlance). The result in Figure 20 is exactly two flagged executions. Each figure corresponds to one direction of the total ordering between b and d. We will inspect both to demonstrate that they are only forbidden as consequence of the total order, and thus if the total order was taken away they would both be allowed.

In Figure 20a note the following. All of the cycles for any kind of edge involve $\mathsf{a} \xrightarrow{\text{po-loc}} \mathsf{b}$ which is not in ob. This means we can avoid a cycle in ob. Recall that if there were a cycle in ob then when we use ob for the other side of the total coherence order it would be a cycle regardless of the total coherence order. Also, note that if we take away $\mathsf{b} \xrightarrow{\text{co}} \mathsf{d}$ (blue) it will also remove $\mathsf{e} \xrightarrow{\text{fr}} \mathsf{d}$. Thus, if we take away $\mathsf{b} \xrightarrow{\text{co}} \mathsf{d}$ by removing the total coherence order of the model, there is no cycle left in the graph and the execution would be allowed.

In Figure 20b note the following. All of the cycles for any kind of edge involve $\mathsf{c} \xrightarrow{\text{bob}} \mathsf{d}$ which is not in `po-loc | ca | rf`. This means we can't establish a cycle in `po-loc | ca | rf`. Thus, if we take away $\mathsf{d} \xrightarrow{\text{co}} \mathsf{b}$ by removing the total coherence order of the model, there is no cycle in ob left in the graph and the execution would be allowed.



(a) `po-loc | ca | rf` Cycle (b) ob Cycle

Fig. 20. Two Flagged Executions