Purr-spectives on Graphs

Jade Alglave[1,2], Artem Khyzha[1], Luc Maranget[3], Nikos Nikoleris[1], Hadrien Renaud[1,2] and Filippo Sestini[1]

[1]Arm Cambridge, United Kingdom
[2]University College London, United Kingdom
[3]INRIA Paris, France

Abstract

The Arm Architecture Formal Team looks after a number of Arm Architecture Representations, amongst which the Arm Concurrency Model. The Arm Concurrency Model is a formal artefact, written in a domain-specific language called cat, which determines which executions of a given program written in Arm assembly are Allowed or Forbidden. The Arm Concurrency Model is also an executable artefact, implemented in a tool called herd7. The herd7 tool displays the executions of a given program as graphs, and this is what leads us here.

We are delighted to have the opportunity to expose a little bit of our work, enough we hope to list a series of practical graph-related challenges that we face in our work. It may well be that some of our challenges have evident, perhaps even easy solutions from an expert point of view.

That is precisely the point: with this paper, we hope to elicit conversations with experts, and perhaps fruitful collaborations, towards solving our challenges and hopefully beyond.

Keywords

Arm Architecture, Arm Architecture Formal Team, Arm Concurrency Model, Arm Assembly, "cat" Language, Litmus Tests, herd7 Tool, Allowed and Forbidden Executions, Abstraction Levels, User-level, Virtual Memory, Instruction Semantics, Graphs, Concurrency, Acyclicity, Irreflexivity, Visualisation, Domain-specific languages, Semantics, Model-driven engineering

CEUR-WS.org/Vol-4338/GCM-invited2.pdf

1. Introduction

Arm devises, maintains and publishes specifications. Specifications allow a user to answer questions about Arm systems such as telephones, tablets, laptops, servers...

An example of a question about the behaviour of an Arm system is a *litmus test*: a small concurrent program written in Arm assembly, which asks whether a certain behaviour is Allowed or Forbidden on Arm hardware.

```
AArch64 MP
{
0:X1=x; 0:X3=y;
1:X3=y; 1:X1=x;
}
 P0           | P1           ;
 MOV W0,#1    | LDR W4,[X3]  ;
 STR W0,[X1]  | LDR W5,[X1]  ;
 MOV W2,#1    |              ;
 STR W2,[X3]  |              ;
exists (1:X4=1 /\ 1:X5=0)
```

Figure 1: An example of a question about the behaviour of an Arm system: the MP litmus test

Figure 1 gives a litmus test called MP, for Message Passing. The test asks what happens if two threads P0 and P1 communicate via shared memory locations x and y, as follows:

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29–July 3, 2026.
jade.alglave@arm.com (J. Alglave); artem.khyzha@arm.com (A. Khyzha); luc.maranget@inria.fr (L. Maranget); nikos.nikoleris@arm.com (N. Nikoleris); hadrien.renaud2@arm.com (H. Renaud); filippo.sestini@arm.com (F. Sestini)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-26

- P0 stores value 1 to x, and sets the flag y with another store.
- P1 loads the flag y and loads x.

If P1 sees the flag set, can it still see the old value of x?

The answer is yes: the Arm Architecture allows P1 to see the old value of x even after seeing the update of the flag y. This could be, for example, because the two stores on P0 have been reordered, or the two loads on P1 have been reordered.

The way the Arm Architecture determines that the MP behaviour is Allowed uses graphs. This paper aims to present an overview of this formalisation, and a number of practical challenges we face in our work. We would welcome comments, feedback, advice and help on any of these topics.

The Arm Architecture Formal Team looks after a number of artefacts distributed by Arm, more precisely the Arm Architecture and Technology Group (ATG). Amongst other things, we look after the Arm Concurrency Model.

2. The Arm Concurrency Model

At a high-level, the Arm Concurrency Model determines how a concurrent program written in Arm AArch64 assembly is Allowed to behave, and in turn, whether Arm hardware is allowed to exhibit that behaviour or not. Figure 1 shows an example of a concurrent behaviour, given as a litmus test.

To forbid this behaviour, one can use, for example, barrier instructions as shown in Figure 2.

```
AArch64 MP+DMB.ST+DMB.LD
{
0:X1=x; 0:X3=y;
1:X3=y; 1:X1=x;
}
 P0           | P1           ;
 MOV W0,#1    | LDR W4,[X3]  ;
 STR W0,[X1]  | DMB LD       ;
 DMB ST       | LDR W5,[X1]  ;
 MOV W2,#1    |              ;
 STR W2,[X3]  |              ;
exists (1:X4=1 /\ 1:X5=0)
```

Figure 2: The MP+DMB.ST+DMB.LD litmus test

There, the DMB ST barrier instruction on P0 ensures that the two stores cannot be reordered, and likewise the DMB LD barrier instruction on P1 ensures that the two loads cannot be reordered.

2.1. Execution graphs

The Concurrency Model determines which behaviour is Allowed (MP) or Forbidden (MP+DMB.ST+DMB.LD). It does so by examining executions, which are given as graphs over Effects. In first approximation, Effects correspond to accesses to memory, for example reading from or writing to memory. Let's look at the graph for the MP behaviour, given in Figure 3.

This test asks whether the following execution is architecturally Allowed:

- The first Store instruction in program order on P0, i.e. STR W0,[X1], generates an Explicit Memory Write Effect of Location x with value 1;
- The second Store instruction in program order on P0, i.e. STR W2,[X3], generates an Explicit Memory Write Effect of Location y with value 1;
- The first Load instruction in program order on P1, i.e. LDR W4,[X3], generates an Explicit Memory Read Effect of Location y;

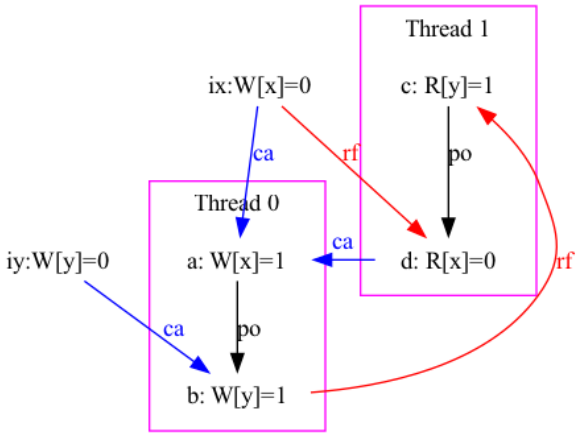


Figure 3: Allowed execution of the MP test

- The second Load instruction in program order on P1, i.e. LDR W5,[X1], generates an Explicit Memory Read Effect of Location x;
- The Register X4 on P1 holds the value 1 in the end, which means that the Explicit Memory Read Effect of Location y on P1 generated by LDR W4,[X3] Reads-from the Explicit Memory Write Effect of Location y on P0 generated by STR W2,[X3];
- The Register X5 on P1 holds the value 0 in the end, which means that the Explicit Memory Read Effect of Location x on P1 generated by LDR W5,[X1] Reads-from the initial state, and therefore is Coherence-before the Explicit Memory Write Effect of Location x on P0 generated by STR W0,[X1].

Let's also look at the graph for the MP+DMB.ST+DMB.LD behaviour, given in Figure 4.

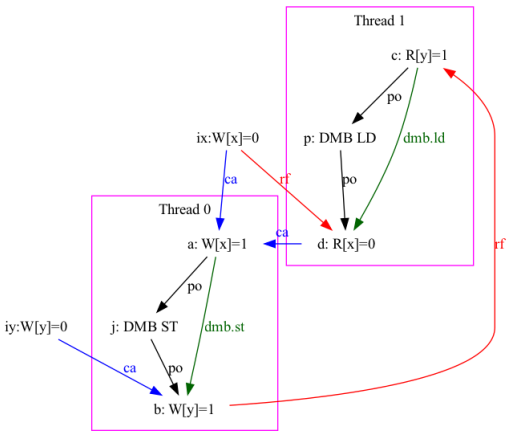


Figure 4: A Forbidden execution of MP+DMB.ST+DMB.LD

The Forbidden execution of MP+DMB.ST+DMB.LD is very similar to the Allowed execution of MP at first glance. But the Forbidden execution of MP+DMB.ST+DMB.LD has a number of differences:

- on P0, a Barrier Effect j:DMB ST, which corresponds to the eponymous instruction in the original MP+DMB.ST+DMB.LD litmus test;
- on P1, a Barrier Effect p: DMB LD, which corresponds to the eponymous instruction in the original MP+DMB.ST+DMB.LD litmus test;
- on P0, an arrow labelled dmb.st between the two Write Effects a:W[x]=1 and b:W[y]=1;
- on P1, an arrow labelled dmb.ld between the two Read Effects c: R[y]=1 and R[x]=0.

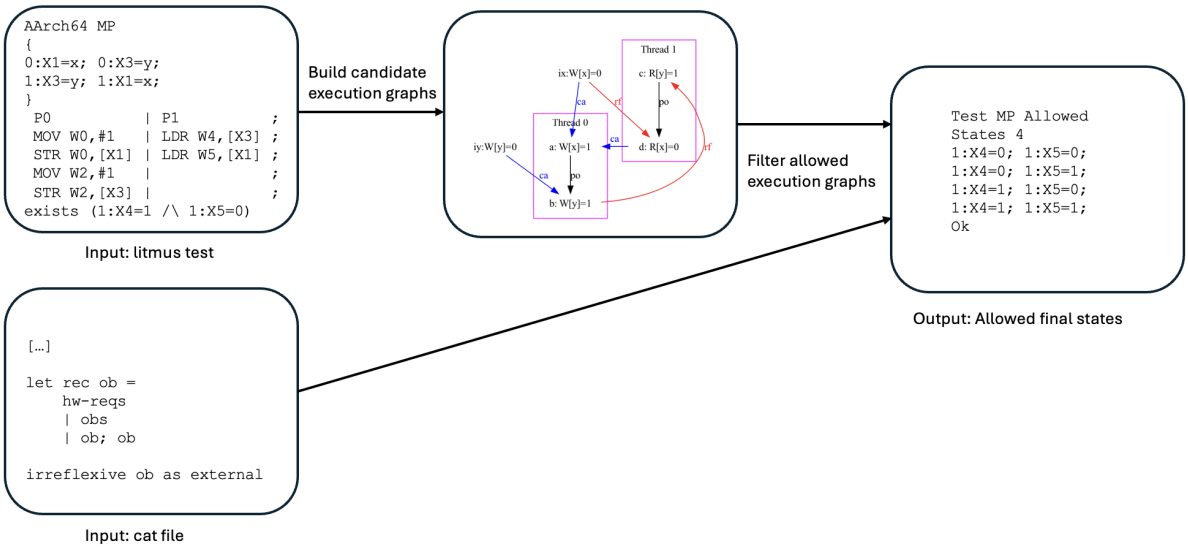


Figure 5: The herd7 tool determines whether a given litmus test is Allowed or Forbidden by a given cat file

The two arrows are what makes all the difference, and forbid the MP+DMB.ST+DMB.LD behaviour when the MP one was allowed. Indeed they intuitively prevent the reorderings of the two Writes on P0 and the two Reads on P1, or in other words they prevent the causes that could lead to the MP behaviour.

2.2. The herd7 tool and the cat language

To help us determine, for each litmus test, whether its behaviour is Allowed or Forbidden, we maintain and enhance the herd7 tool [1]. This tool takes as input a litmus test and a *cat file*, and decides whether the behaviour of the litmus test is Allowed by the cat file, as illustrated in Figure 5.

The cat language is a domain-specific language for describing concurrency models [2], which has a formal syntax and semantics [3], and is implemented in particular in the herd7 tool [1]. A cat file is a series of definitions of relations over Effects. Primitives include relations such as program-order, Reads-from or Coherence-after, and a cat file defines combinations thereof, such as union, intersection, transitive closure. A cat file also gives a number of axioms, or rules, about these definitions. One can for example require a given relation to be acyclic, or non-empty.

When given a litmus test such as MP, herd7 will build execution graphs such as the one given in Figure 3, then check whether these graphs respect the rules given in the cat file. When a graph respects the cat rules, the behaviour is Allowed; otherwise it is Forbidden.

In the Arm Concurrency Model for example, we define an overarching relation called Ordered-before, as the transitive closure of the union of two other relations called Hardware-requirements (intuitively the local ordering requirements per thread), and Observed-by (intuitively the communications or observations between threads). In cat, this is written as:

```
let rec ob =
    hw-reqs
    | obs
    | ob; ob
```

Here we see a number of cat constructs:

- the union of two relations R1 and R2: "`R1 | R2`";
- the sequence "`R1 ; R2`";
- the recursive definition "`let rec def = ... def`".

This transliterates to [4, B2.3.9]:

An Effect E1 is Ordered-before an Effect E2 if one of the following applies:

- *E1 is Hardware-required-ordered-before E2.*
- *E1 is Observed-by E2.*
- *All of the following apply:*
  - *E1 is Ordered-before E3.*
  - *E3 is Ordered-before E2.*

The Ordered-before relation is required to be acyclic, or more strictly, since Ordered-before is transitive, it is required to be irreflexive. Intuitively, a cycle in Ordered-before creates some sort of "causal loop" which is not permitted. In cat, this is:

```
irreflexive ob as external
```

Here we see the cat constructs:

- "`irreflexive R`" demands that the relation R is irreflexive;
- "`rule as name`" gives a name to a rule.

This requirement is called the External Visibility Requirement, and transliterates as follows [4, B2.3.9]:

an Architecturally Allowed Execution must not exhibit a cycle in the Ordered-before relation.

2.3. Instruction semantics

To go from a litmus test such as MP (resp. MP+DMB.ST+DMB.LD) to a graph such as the one given in Figure 3 (resp. Figure 4), we need something that goes from the Arm AArch64 assembly syntax to the Effects and relations to which the cat file applies. In Figure 5, this is the step that is labelled "Build candidate execution graphs".

To do this, we need a number of stepping stones:

- for each instruction in the program, such as STR or LDR in the MP litmus test, we need a semantics of that instruction in terms of the Effects that this instruction generates, e.g. how it affects memory;
- for each thread in the program, such as P0 and P1 in MP, we need to stitch together these per-instruction semantics into a combined semantics for each thread;
- for the whole program, we need to enumerate the possible interactions across threads: for example in the execution of MP given in Figure 3, the Read of y on P1 taking its value from the Write of y on P0 is one such possible interaction (but equally they could be the other way around); the Read of x on P1 being overwritten by the Write of x on P0 is another possible interaction (but again equally they could be the other way around).

2.3.1. Operators for building graphs

The semantics of each instruction is a mini-graph. We can already see this in Figures 3 and 4:

- the semantics of a store instruction (e.g. for MP: STR W0,[X1] on P0) is given in the graph as a Memory Write Effect (e.g. a:W[x]=1);
- the semantics of a load instruction (e.g. for MP: LDR W4,[X3] on P1) is given in the graph as a Memory Read Effect (e.g. c:R[y]=1);
- the semantics of a barrier instruction (e.g. for MP+DMB.ST+DMB.LD: DMB ST on P0) is given in the graph as a Barrier Effect (i.e. j:DMB ST).

These Effects are an abstraction of the full semantics of these instructions. For example, a (still abstract but more complete) semantics of STR W0,[X1] and LDR W4,[X3] includes their operation over the registers that constitute their arguments: Figure 6 gives their user-level semantics side by side.

To build these graphs, we use a number of operators, which in herd7 are used in `herd/AArch64Sem.ml` [5] to define the semantics of each instruction.

Two important operators are »= and »|, which one can then use to assemble the semantics of a given instruction (see e.g. `build_semantics` in `herd/AArch64Sem`).

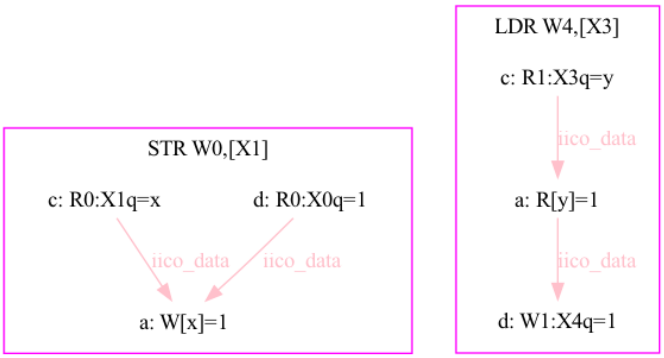


Figure 6: User-level semantics of STR W0,[X1] and LDR W4,[X3]

Operator »= Intuitively a `»=` b means "do a, collect its result and feed it into b". A concrete example would be:

```
read_reg rs >>= fun d -> write_mem x d
```

which reads the register rs, gets some data d, and writes that data d back to memory location x.

In the user-level semantics graph in Figure 6, this is what creates the two pink "iico_data" arrows:

- both start from a register read (the Register Read of X1 labelled c and the Register Read of X0 labelled d respectively);
- from which we get some data (x and 1 respectively);
- both data then get passed on the Memory Write Effect labelled a.

Similarly for the user-level semantics graph of LDR:

- the first arrow, from c to a, uses »= to transmit the Location y;
- the second arrow, from a to d, uses »= to transmit the value 1 around.

Operator »| Intuitively a `»|` b means "do a in parallel with b". This is for cases when we know that certain accesses do not need to be ordered. For example:

```
(read_reg rs >>| read_reg rd) >>= fun (v,x) -> write_mem x v
```

which reads the register rs and in parallel reads the register rd, gets a value v from rs and a location x from rd, and writes that value v back to that memory location x.

This is what happens in the semantics of STR: the Register Reads c and d do not need to be ordered (unlike the Reads of LDR), and we can use `»|` to build the semantics of STR.

2.3.2. Levels of abstraction

Each instruction can be seen at different levels of abstraction. We have already seen this with e.g LDR:

- as a single Memory Read Effect in Figure 3;
- as a Memory Read Effect accompanied with Register Effects in Figure 6.

But this assumes that an instruction (for example an LDR) directly accesses memory, after having obtained the name of the location from one of its register arguments. In practice, we cannot access memory directly, and instead interact with it through virtual addresses. Virtual addresses are mapped to physical locations through page tables, and each entry in a page table indicates at a minimum whether it is valid or not. When the entry is invalid, we raise what is called an exception.

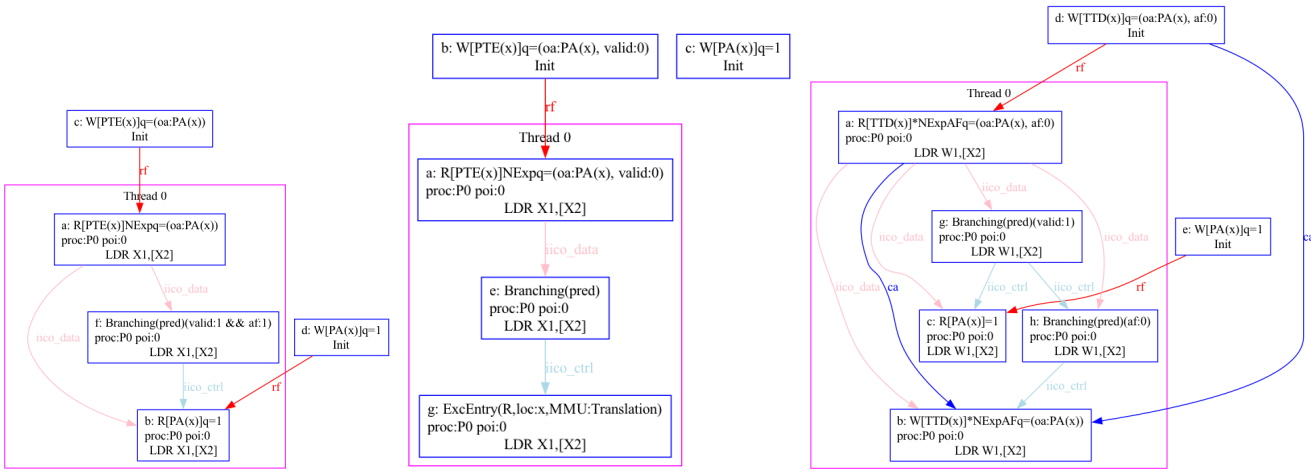


Figure 7: Aspects of the virtual memory semantics of LDR

So we can see that there is another level of abstraction, where we account for the existence of virtual addresses and page tables. At that level, the instruction semantics graphs become more complex: we give illustrations for LDR in Figure 7.

There we can see that the semantics of LDR also features Implicit Effects, for example the Read labelled a. This Read accesses the page table entry for the virtual address x, and depending on whether the entry is valid or not, will either raise an Exception (Effect g, middle drawing) or access memory as before (Effect b, left-hand-side drawing).

3. A number of practical challenges we face

With this background, we can now list a number of practical challenges we face.

3.1. Visualisation of execution graphs

The execution graphs shown throughout the paper are produced by herd7 as graphviz/DOT files. By setting the appropriate command line arguments, herd7 can be instructed to represent the execution graphs it generates internally as graphviz/DOT files. In many respects, this is a convenient choice, since DOT is a standard, well-established text-based graph description language, with command line tools to render its graphs into various formats.

But this approach also has an important limitation, namely that DOT is used both as the format emitted by herd7 and the visual description consumed by the DOT renderer. As a result, by the time the user sees the picture, many visualisation decisions have already been compiled into a static dot file and are very difficult to change after the fact. The user can try to generate different graphs by passing different command line arguments to herd7, or by editing the dot file by hand, but neither workflow is particularly convenient, or robust.

One example of this is when one wants to fine-tune the position of nodes and edges in a herd7-generated graph: herd7 currently does not offer ways to control this, and modifying the dot files to affect the position of nodes and edges is difficult. This layout problem is not merely aesthetic: when the dot engine chooses an unhelpful node placement or edge routing, this can make it harder to follow the relations, paths, or cycles that the graph is meant to explain, and which are rather crucial to our work.

The decision of which edges to draw in the graph also raises some tension between visual readability and semantic completeness of the rendered picture. For example, herd7 will by default omit transitive edges from the DOT graph unless explicitly requested by the user. This is usually the right default in terms of readability. On the other hand, when trying to visualise a cycle in a transitive relation, it may be important to keep those transitive edges in the picture, as discussed in Section 3.9.

More generally, what constitutes a useful visual projection of an execution graph into a picture is not fixed once and for all, but depends on what particular aspect of the execution the user is interested in exploring at that particular moment. We discuss some of these needs in Section 3.8. Ideally, users would be able to reveal or hide different parts of the visualised graph interactively, or at least through rapid iteration, without having to ask herd7 to serialise a new DOT file each time.

Thus, the issue is not simply that DOT rendering engines may produce unhelpful layouts. Rather, it is that herd7 currently emits a static artefact that fixes a particular view of a generated execution graph, instead of a semantic graph object that can be explored through different views.

Perhaps one solution could be to keep DOT as the format emitted by herd7 (perhaps with a richer set of attributes), but use a more capable, and possibly more interactive renderer. However, we also wonder whether an alternative to DOT exists, that constitutes a more semantic intermediate graph representation. Such a format would describe the events, relations, cycles, and any other aspect of an execution graph in full detail, without committing to how it must be drawn. It would then support both interactive visualisation and rendering to static image formats. Ideally, this would be an established domain-specific language rather than our own custom format.

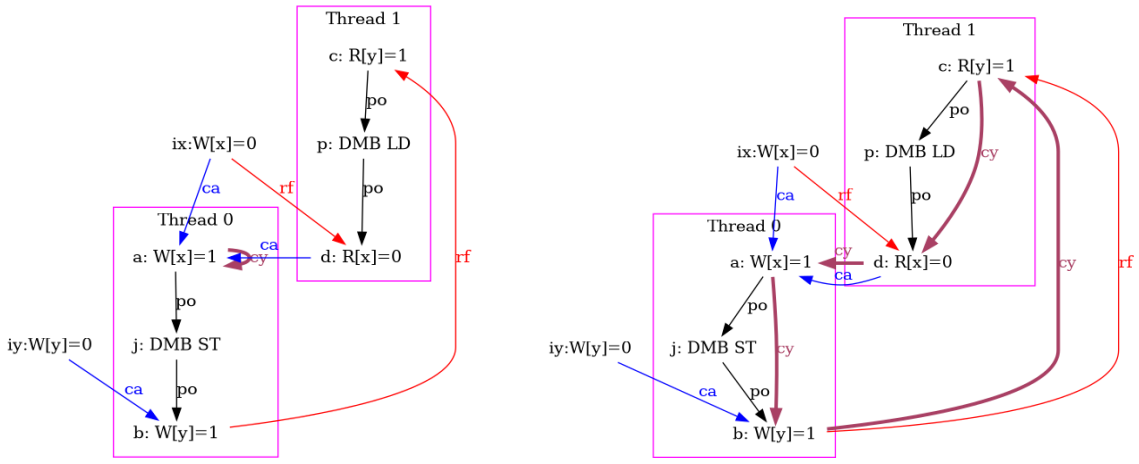
3.2. Visualising irreflexivity violations meaningfully

When a litmus test is Forbidden by a cat file, there are two questions which become relevant:

- What cat requirements are not satisfied, and
- What paths in the Forbidden execution violate these requirements.

Identifying the paths of an execution graph that violate a given requirement is still a challenge. Indeed, many cat requirements are encoded as `irreflexive` checks in the cat file. The definition of irreflexivity entails that visualising the path that violates the check is not very helpful.

Consider for example the graphs given in Figure 8.



(a) Irreflexivity test. (b) Acyclicity test.

Figure 8: Illustrating why the execution of the test MP+DMB.ST+DMB.LD (given in Figure 2) is Forbidden.

On the left, we see a failure of the External Requirement phrased as an irreflexivity test, the current idiomatic phrasing. On the right, we see a failure of the External Requirement, rephrased as an acyclicity test, which is an equivalent phrasing since the Ordered-before relation is transitive.

Of the two diagrams, the right-hand-side one is the most helpful: it gives a clear path throughout the Forbidden execution, which explains why the execution violates the cat file.

It could be that we only need to swap irreflexivity checks for acyclicity checks under the hood, leaving the appearance of the cat file untouched. But this is not necessarily an anodine swap from a performance point of view, a question which we discuss next.

3.3. Data structures and algorithms

Our cat interpreter operates on relations that can be seen as graphs. We did not rely on graph libraries such as ocamlgraph [6] in our initial prototype, because we wanted to minimise software dependencies and our needs seemed mundane: a few elementary algorithms on persistent, unlabeled, graphs. Indeed we mostly needed to implement the operators of the cat language: union, intersection, inclusion, ... To that aim, representing graphs as sets of pairs, using OCaml `Set` library looked sufficient: e.g. computing the union `r|s` relies on OCaml union of balanced trees, which is both efficient and well tested.

But at least one outlier made us reconsider our initial stance. Indeed the sequence operator `r;s` cannot be implemented efficiently on sets of pairs, because it requires efficient access to the successors in `s` of each member of the range of `r`. Hence we relied on a three step process:

1. represent graphs as maps from nodes to the set of their successors;
2. perform the sequence operation by a traversal of `r`; and
3. translate the result back to a set of pairs.

In the end, we realised that the OCaml standard library module `Map`—whose implementation builds on the same balanced tree structure as the module `Set`— provides a function `merge` that is general enough to implement union, intersection, etc. of maps and that is just as convenient and efficient as its counterparts on sets.

Experience shows that our most problematic operation is the transitive closure `r0+`. Historically we relied on a naive implementation inspired by the equivalence with the recursive declaration:

```
let rec r = r0 | r;r
```

That is, we iterated the sequence until fixpoint. Having identified this to be a bottleneck, we worked to replace our naive original implementation with an implementation of the more sophisticated algorithm of Nuutila [7]. Nuutila's algorithm builds upon Tarjan's algorithm [8] for computing strongly connected components.

We note that we use Tarjan's algorithm elsewhere in the code base also, in a very different context: for herd7's solver to handle equations according to their dependencies. Using the exact same code for two different purposes on very different graphs is a tribute to functional programming and OCaml module system. Avoiding duplication of code especially when complex and error prone is not only a matter of aesthetics, as code sharing reinforces reliability and facilitates maintenance.

3.4. Enhancing performance by combining operators

To improve the performance of the cat interpreter, one may combine operators so as to avoid intermediate computations and structures. We first illustrate these suggestions. The AArch64 cat model contains expressions such as:

```
[Exp & M]; rf & ext; [Exp & M]
```

The phrase "`rf & ext`" builds a restriction of the Reads-from relation (rf), so that the extremities of the resulting relations are on different threads (ext). The set "`[Exp & M]`" restricts the set of all Memory Effects (M) to only the Explicit Effects (Exp), i.e. intuitively the ones that the user expects to see performed: a Read of Memory when doing a load, a Write of Memory when doing a store, abstracting away e.g. address translation considerations.

But to understand the point of this section, the exact meaning of sets (`Exp`, `M`) and relations (`rf`, `ext`) involved is not important. We focus on operators, emphasising that they perform computations,

- Computing the intersections of sets `Exp & M` and of relations `rf & ext` yields the sets s and the relation r, respectively.
- The operator `[`s`]` yields the identity on set s, written id_s.
- Finally we compute the sequence of three relations $\mathsf{id}_s ; r ; \mathsf{id}_s$.

Once we have observed that $[s];r$ (resp. $r;[s]$) is the relation r whose domain (resp. range) has been intersected with the set s, we see that we do not need to compute both identities *and* sequences, and some computation can be spared. Moreover, due to the asymmetry of the map representation in OCaml, it is more rewarding to compute sequences left-to-right. Here, compute $(\mathsf{id}_s;r);\mathsf{id}_s$ rather than $\mathsf{id}_s;(r;\mathsf{id}_s)$.

A similar approach is possible to avoid transitive closures, a target of interest given the cost of this operation. We can start with a few observations.

Some transitive closure operations are easy to spare, while evaluating expressions. For instance, we can compute $(r+)+$ as $r+$ or, perhaps more significantly in practice, $(\ldots|r+|\ldots)+$ as $(\ldots|r|\ldots)+$. We may not be aware of other similar situations and would welcome help to find more.

However, the cat models seldom include long expressions involving nested transitive closures. Instead, we often have the following, semantically equivalent, sequence of definitions:

```
let r1 = r0+
let r2 = (...|r1|...)+
```

And most often, the transitive closure of transitive closure is even hidden by syntax choices such as:

```
let rec r1 = r0 | (r1;r1)
let rec r2 = ... | r1 | ... | (r2;r2)
```

A simple syntactic analysis can assert that the recursive definitions `r1` and `r2` are equivalent to the non-recursive ones above.

A similar situation occurs with model requirements. The following requirements:

```
irreflexive r1+ as check1
acyclic (...|r2+|...) as check2
```

Can be replaced by the less costly:

```
acyclic r1 as check1
acyclic (...|r2|...) as check2
```

In the case of `check1` we trade an irreflexivity check and a transitive closure for an acyclicity check, a likely beneficial operation. In the case of `check2` we avoid a transitive closure.

Achieving performance gains from the previous observations is not straightforward: there is a trade-off to be settled between the gains to be expected against the cost of the static analysis and additional interpreter actions and structures involved. As an example of such an additional cost, one can design a new internal representation of the value of $r+$ as a pair of a delayed computation and of an "implicit" transitive closure of r, leveraging OCaml lazy evaluation support to avoid useless computations and re-computations of the first item of this pair. The appropriate first or second item would then be selected depending on context. Obviously, replacing simple relation values by this sophisticated data structure would incur costs, such as additional memory occupancy and tests.

3.5. Algorithmic gains

It is highly likely that we have not selected the best algorithms available for some frequent operations, for instance acyclicity checks or, again, transitive closure. Here we would benefit from informed expertise.

Moreover, we suspect that our graphs are not random: they are usually sparse with the exception of unions of total orders (which we use to represent e.g. program order). Maps of nodes to the set of their successors are an adequate implementation, whilst an implementation as ordered lists would yield important gains as regards memory occupancy.

Similarly, some operations can probably be optimised by selecting different algorithms depending on the nature of their input. Implementing such a variety of representations and algorithms efficiently will prove challenging. Another possibility may be to attach metrics to graphs, again with the same objective of adapting algorithms to their input. Choosing relevant metrics remains an open question for us and we would once again much welcome expert input and contributions.

3.6. Rationalising operators for building graphs

We have seen above that to define instruction semantics, we use some operators to combine the graphs.

In practice, to write instruction semantics, we interview stakeholders such as our colleagues, architects, hardware designers and software developers. We draw our understanding of their intent, and set out to implement that using our operators.

In an ideal world, we would be able to readily go from the target graph as discussed with our stakeholder to its implementation using our operators. In practice this is often very hard. We would welcome input and contributions that would allow us to go automatically from a candidate instruction semantics graph to its mapping to our set of operators, or at least a close approximation thereof.

Relatedly, the set of operators would ideally be a proper language itself, a domain-specific language for building instruction semantics graphs, with a formal semantics. The number of operators would be minimal, and we would not have to produce specialised versions of them in arduous cases, unless the case was genuinely a special one.

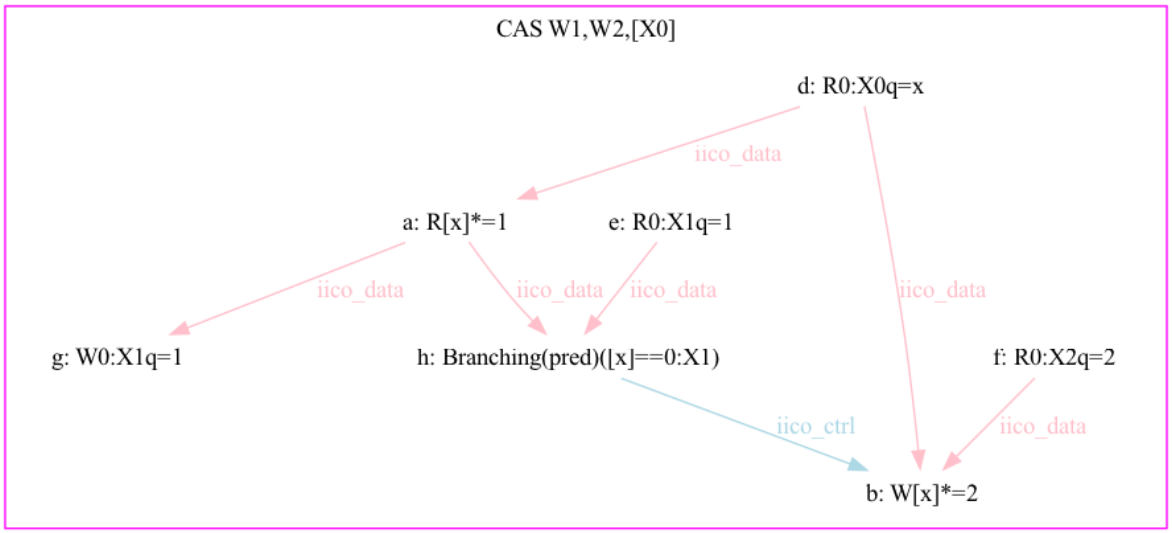


Figure 9: User-level semantics of CAS W1,W2,[X0]

Currently, a number of instructions escape our core set of operators, and for these we have to build specialised, dedicated operators. We would welcome insights as to whether this is an inherent difficulty, or if we could adjust our core set of operators to avoid such a situation.

As an illustration, consider one of the instruction semantics graphs of the Compare-And-Swap (CAS) instruction displayed in Figure 9, as elaborated from discussions with stakeholders. In this diagram:

- The Register Read Effect d:R0:X0=x gets Memory Location x from Register X0;
- The Register Read Effect e:R0:X1=1 gets the value 1 from Register X1;
- The Register Read Effect f:R0:X2=2 gets the value 2 from Register X2;
- The Explicit Memory Read a:R[x]=1 gets the value 1 from Memory Location x;
- The Branching Effect acknowledges the fact that the value of x and the value of X1 are equal, and therefore the swap can occur;
- The Register Write Effect g:W0:X1=1 writes the value 1 (read from x) back to Register X1;
- The Memory Write Effect b:W[x]=2 writes the value 2 to Memory Location x.

The iico_data arrows represent Intrinsic Data Dependencies, which correspond intuitively to situations where some data is transmitted from one Effect to another, such as a value or a Location.

The CAS semantics features another type of Intrinsic Dependency, labelled iico_ctrl. This type of dependency relates to the control flow within an instruction, in this case, whether the value 2 is written

to the memory location x depends upon the result of the comparison of the old value of x with the value 1.

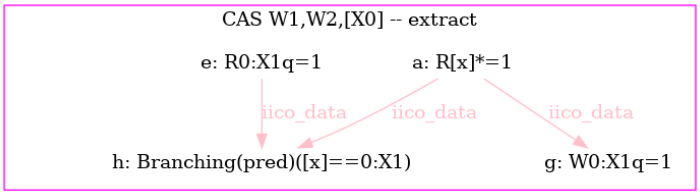


Figure 10: A graph that cannot be built with the operators of Section 2.3.1

The relative complexity of the CAS semantics shape in particular extends beyond the reach of the simple operators of section 2.3.1. Consider for example a subgraph of the CAS semantics graph in Figure 9, i.e. the graph of Figure 10.

The way in which the Effects a and e relate to the Effect h is akin to the semantics of STR in Figure 6, and indeed we can relate the Effects a and e to the Effect h using the operator »| as we do for STR.

But we cannot seem to find a way to introduce the Effect g in the picture without a specialised operator. The difficulty there is that the Effect g is related to the Effect a, but not to the Effect b.

We could adapt our operators, for example »=, into a sibling operator written $\gg=_1$. The graph of Figure 10 would then be constructed as:

$$(e \gg| (a \gg=_1 g)) \gg= h$$

Alternatively, we could invent a four argument operator, to build the graph of Figure 10 directly.

We would welcome insights as to how to best design a robust collection of operators, from which we could build all the shapes we are aware of, including "difficult" ones such as CAS, and hopefully beyond.

3.7. Managing collections of litmus tests as collections of forbidden cycles

As we continuously extend the Arm Concurrency Model in depth (with more features like virtual memory [9]) or in breadth (with more instructions supported), many litmus tests are composed to study potential ordering implications of the extensions. We maintain a catalogue, a growing collection of such litmus tests [10], and tool support based on identifying reasons for why a given litmus test is forbidden can facilitate interpreting and searching through the collection of tests.

For example, developing a procedure for comparing two litmus tests based on forbidden cycles present in their execution graphs would allow to identify litmus tests in the catalogue that, despite being different literally, for instance, due to different amount of instructions, are nevertheless identical in terms of forbidden cycles.

Taking this idea further, one could envision situations where a big litmus test can be decomposed into several smaller litmus tests, each disjointly illustrating cycles present in executions of a bigger litmus test. For instance, consider the CopyOnWrite litmus test in Figure 11. This litmus test illustrates a principle of virtual memory abstraction, with the idea that threads execution on P0 and P1 correspond to application-level code, while P2 corresponds to the code of a hypervisor that affects how virtual addresses used on P0 and P1 are mapped to physical addresses, but in a way that does not invalidate the concurrency semantics expected by P0 and P1. Naturally, the set of graphs of forbidden executions for this test will include graphs of two smaller litmus tests:

- CoRR in Figure 12, which corresponds to the threads P0 and P1;
- CoRR-TTD in Figure 13, which corresponds to the thread P2 happening concurrently with address translations occurring on P0.

A tool capable of identifying all forbidden cycles in the execution graphs of a given litmus test could attempt to determine whether two or more of smaller litmus tests together produce a set of forbidden cycles subsuming those in a given original large litmus test.

Moreover, the aforementioned procedures could be further enhanced through approximate comparison of two forbidden cycles. A notion of similarity of two cycles could be devised based on the definitions of relations given by a cat file. Thus, for instance, in principle a cycle could feature two or more edges of the same type (e.g. an address dependency), corresponding to a chain of that type of ordering between instructions. While such a chain is not redundant by definition, it may not necessarily illustrate a novel ordering principle in comparison to a single edge of that type. Cycles could be considered similar in regards to that.

```
AArch64 CopyOnWrite
(*Forbidden*)
{
int x=1; int y= 0; int z= 0;
[TTD(z)] = (oa:PA(x)); 2:X7=(oa:PA(x),valid:0);
2:X6=TTD(x); 2:X8=TTD(y);
0:X2=x; 2:X2=x; 2:X4=y; 2:X10=z; 1:X2=x;
}
P0           | P1            | P2              ;
                             (*invalidate VA x*)
             |               | STR X7,[X6]     ;
                             (*TLB maintenance*)
             |               | LSR X3,X2,12    ;
             |               | DSB ISH         ;
             |               | TLBI VAAE1IS,X3 ;
             |               | DSB ISH         ;
                             (*memcopy via VA z*)
             |               | LDR X5,[X10]    ;
             |               | STR X5,[X4]     ;
             |               | DMB ISH         ;
                             (*update TTD of x*)
LDR X0,[X2] | MOV X0,2      | LDR X1,[X8]     ;
LDR X1,[X2] | STR X0,[X2]   | STR X1,[X6]     ;
exists 0:X0=2 /\ 0:X1=1
```

Figure 11: The CopyOnWrite litmus test

```
AArch64 coRR
{
0:X2=x; 1:X2=x;
int64_t x=1;
}
P0           | P1          ;
LDR X0,[X2] | MOV X0,#2   ;
LDR X1,[X2] | STR X0,[X2];
exists(0:X0=2 /\ 0:X1=1)
```

Figure 12: The coRR litmus test

3.8. Zooming in or out of instruction semantics levels of abstraction

Semantics of a given instruction has different levels of abstraction. When executed on modern CPUs, an instruction in a program goes through stages, many of which may result in memory accesses. Thus, one could expect a modern CPU to take the address of the next instruction to execute from the program counter, and then:

```
AArch64 coRR-TTD
{
y=1; PTE(x)=(oa:PA(x),valid:1);
0:X1=x; 1:X1=x;
1:X5=(oa:PA(x),valid:0);
1:X6=(oa:PA(y),valid:1);
1:X7=PTE(x);
}
P0              | P1              ;
L1:LDR W2,[X1] | STR X5,[X7]     ;
L0:LDR W4,[X1] | DSB ISH         ;
                | LSR X9,X1,#12   ;
                | TLBI VAAE1IS,X9;
                | DSB ISH         ;
                | STR X6,[X7]     ;
exists(0:X2=1/\ 0:X4=0 /\ ~fault(P0:L0,x))
```

Figure 13: The coRR-TTD litmus test

- to read a translation table descriptor in memory
- to translate the virtual address of the next instruction into a physical address;
- to read the instruction by the physical address in memory;
- to execute the instruction, which in the case of instructions accessing memory like LDR or STR may mean extra steps such as:
  - to read a translation table descriptor in memory;
  - to translate the virtual address of the destination of the memory access;
  - to read a memory tag from memory;
  - to check it against the memory tag in the address,
  - to read from the desired address in memory.

All such memory accesses may be subject to concurrency, and therefore need to be considered by the Concurrency Model. However, not all concurrency scenarios involve all types of memory accesses at the same time, as different layers of the software stack tend to be responsible for them.

Our herd7 tool can analyse the code of litmus tests, identify memory accesses and what kind of concurrency they could be subject to, and choose an appropriate level of detail to consider for each of them. But in the general case, a graph for a litmus test may be cluttered and hard to interpret visually.

We would welcome advances that facilitate studying execution graphs assuming different amount of detail to the semantics of executing selected instructions. For example, consider CopyOnWrite litmus test in Figure 11. In principle, the overall set of its execution graphs requires virtual memory semantics for instructions on P0. However, only a subset of the forbidden execution graphs will contain a cycle involving Effects that are due to virtual memory – and such graphs could automatically hide detail irrelevant to understanding the cycle. But we know that none of the forbidden execution graphs will feature cycles involving Registers, and therefore Effects corresponding to Register Reads and Writes could be abstracted away.

Post-processing of the execution graphs that are not forbidden could similarly benefit from varying the level of abstraction for instruction semantics. This could be done by providing input about which semantic aspects are necessary: Register Effects, Explicit Effects, Implicit Effects due to virtual memory....

3.9. Finding and highlighting "the right cycle"

When a litmus test is run with herd7, the output can be given in the form of execution graphs, each of which could be seen as one of the following:

- Architecturally Forbidden, when the graph violates the requirements of the cat specification, typically indicating the presence of a cycle in the graph;
- Architecturally Allowed, otherwise.

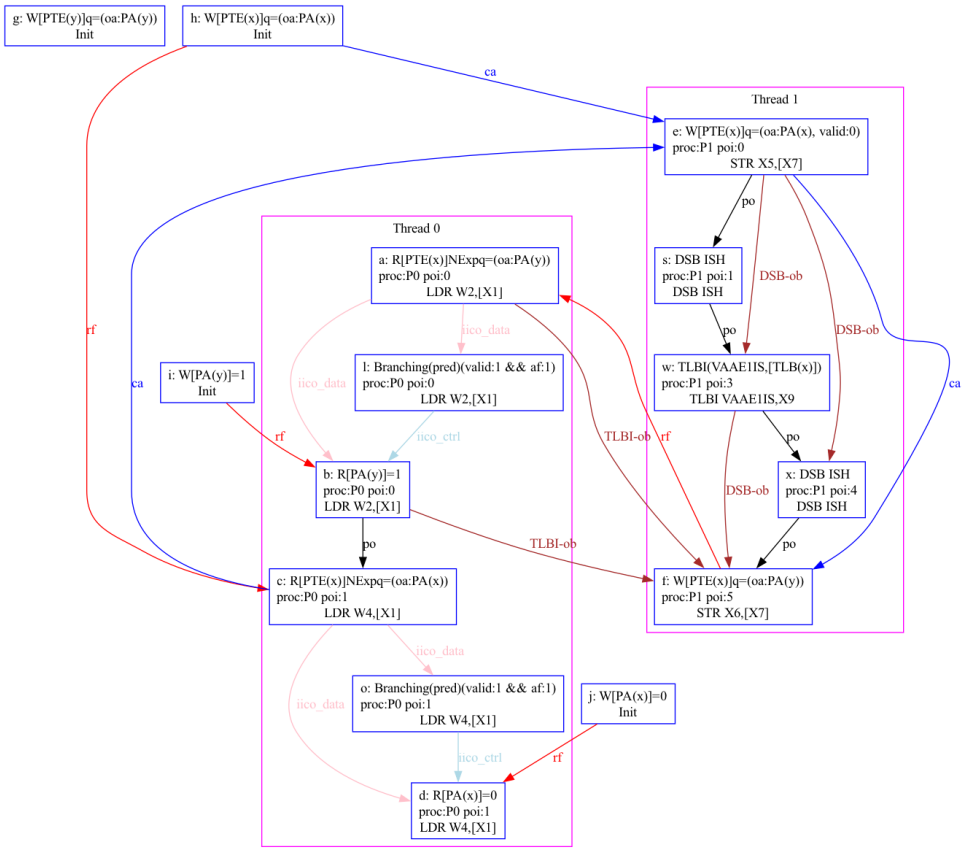


Figure 14: The coRR-TTD litmus test and one of its forbidden executions

Several aspects of our work, e.g. development of the Arm Concurrency Model and its application to concurrency scenarios of interest, involve close studying of Forbidden execution graphs. Therefore any visual aid for displaying most relevant information on graphs is incredibly helpful.

In Section 3.2, we discussed the challenge of displaying the reason for forbidding an execution graph in the form of a cycle rather than a loop. But even once the cycle is displayed, a number of new challenges arise.

Firstly, once the cycle in a relation—most often, the "Ordered-before" relation—is identified and displayed, it is non-trivial to recognise the exact reasons for why Ordered-before contains a cycle. The default mode of displaying a graph in herd7 shows a number of edges that serve as primitives for building other relations of interest, such as: `rf`, `ca`, `po`.

There is, however, quite a gap between those primitives and a cycle in the Ordered-before, as there are multiple levels of auxiliary relations contributing to the definition of Ordered-before, and displaying information from those levels speeds up the analysis of the litmus test.

Thus, for instance, the execution of the `coRR-TTD` litmus test in Figure 14 illustrates the need for the auxiliary relations called `TLBI-ob` and `DSB-ob`.

Presently, there is no way for the user of the tools to quickly understand which auxiliary relations corresponding to the ordering rules need to be displayed to understand the execution graph. The understanding either comes from a prior knowledge or from a guessing game of repeatedly enabling and disabling the visualisation of relations until the helpful one is found.

Once the need for `TLBI-ob` and `DSB-ob` is discovered, we can of course display them with herd7. However, generally speaking, those relations will produce more edges in the graph than required for a

cycle: only the edges between Effects a and f are needed for the cycle in Figure 14, while others clutter the display of the graph.

Additionally, auxiliary relations like `TLBI-ob` can themselves be relatively complex, as they may be built from a disjunction of multiple ordering rules. Knowing which of the disjuncts contributes to the cycle can help with understanding the graph. At the same time, unfolding all the auxiliary definitions used in `TLBI-ob` for the display may be adding too much detail into the display of the cycle, so a degree of user control may be of help.

Overall, finding ways of adding or removing relevant detail to cycles in Ordered-before could simplify analysis of tests. This could include finding ways of annotating the cat specification with indication of which auxiliary relations are the ordering rules to latch default visualisation strategy onto.

4. Conclusion

We have given an introduction to the work done by the Arm Architecture Formal Team in the area of concurrency. More specifically, we look after, maintain and extend the Arm Concurrency Model, a formal artefact which determines which executions of concurrent programs written in Arm assembly are Allowed or Forbidden.

The herd7 tool allows to execute this artefact, written in the domain-specific language cat, so that we can determine automatically whether litmus tests are Allowed or Forbidden.

Allowed and Forbidden executions are represented as graphs; we have shown a few examples of such executions, and why certain ones are Allowed and others are Forbidden.

We have also talked about how we build the mini-graphs which represent Arm instruction semantics, and how such semantics have different levels of abstraction.

Equipped with that background, we have listed a number of genuine practical challenges that we as a team face in our work extending the Arm Concurrency Model and the associated tool support. We have not discriminated the potential or apparent difficulty of the challenges we presented. We hope to interact with experts on any of these challenges, and start conversations and collaborations.

Declaration on Generative AI

The authors did not use any Generative AI tool during preparation of this work.

References

[1] J. Alglave, L. Maranget, herdtools7 (2026). https://github.com/herd/herdtools7.

[2] J. Alglave, L. Maranget, M. Tautschnig, Herding cats, TOPLAS (2014).

[3] J. Alglave, L. Maranget, P. Cousot, Syntax and semantic of cat (2016). https://arxiv.org/abs/1608.07531.

[4] Arm, Arm Architecture Reference Manual for A-Profile Architecture (2026). https://developer.arm.com/documentation/ddi0487/mb/?lang=en.

[5] Various (2026). https://github.com/herd/herdtools7/blob/master/herd/AArch64Sem.ml.

[6] S. Conchon, J.-C. Filliâtre, J. Signoles, Designing a generic graph library using ml functors., Trends in functional programming 8 (2007) 124–140.

[7] E. Nuutila, An efficient transitive closure algorithm for cyclic digraphs, Information Processing Letters 52 (1994) 207–213. doi:10.1016/0020-0190(94)90128-7.

[8] R. Tarjan, Depth-first search and linear graph algorithms, SIAM Journal on Computing 1 (1972) 146–160. doi:10.1137/0201010.

[9] J. Alglave, R. Grisenthwaite, A. Khyzha, L. Maranget, N. Nikoleris, Puss In Boots: On Formalizing Arm's Virtual Memory System Architecture, IEEE MICRO (2024).

[10] Various (2026). https://github.com/herd/herdtools7/tree/master/catalogue.