On the Role of Prose in Specifications

Jade Alglave
Arm, Cambridge, UK
University College London, UK

Abstract

Specifications should let users find answers to their questions. Those answers should be accessible, unambiguous, consensual, reproducible, auditable, and they should be traceable to the artefacts users actually read.

This paper uses work done by the Arm Architecture Formal Team as a case study in the tensions between those requirements.

2012 ACM Subject Classification Theory of computation → Concurrency; Software and its engineering → Formal software verification; Computer systems organization → Architectures

Keywords and phrases Arm Architecture, (prose, formal, executable, queryable, accessible) specifications, concurrency, herdtools, litmus tests, instruction set, Architecture Specification Language (ASL), The Architecture Speaks, query interface

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2026.1

Category Invited Talk

Acknowledgements I would like to thank Nikos Nikoleris, Simon Peyton Jones and Daryl Stewart for taking the time to review various drafts and provide helpful feedback. I would like to give a big shout-out to the Arm Architecture Formal Team who works tiredlessly and with good cheer towards our ambition. I would like to thank Richard Grisenthwaite for giving the team the time and space needed to pursue our work and shape our vision.

1 Introduction

Specifications should let users find answers to their questions. Often, specifications are written in prose, even technical ones. This may have drawbacks; for example, prose may be ambiguous, or less accessible to non-native speakers. There might be a temptation to resolve this by formalising specifications, which at Arm is the original goal of the Architecture Formal Team. Throughout the paper, I use work done by the team as illustration.

For the longest time I saw the prose aspect of our work through a negative lens: often as a burden, something we needed to formalise away and simply would not see the end of before we were all very old indeed; sometimes as a "necessary evil" to communicate formalisations, which invariably would need to be discussed in depth and at length to reach relatively consensual phrasings with stakeholders and be ratified before entering the Arm Architecture Reference Manual.

Somewhat recently, my stance on prose has changed and I try to expose why, and how the team's work has evolved under that renewed steer. For the Arm Architecture Formal Team, prose cannot be left outside specification engineering: prose is the main communication medium for our stakeholders.

Hence for us prose cannot be replaced by formal models, even executable ones. Instead prose needs to be engineered: connected to formal artefacts, generated where appropriate, queried by tools, tested through behavioural questions, and treated as a first-class specification artefact. Relatedly certain specification aspects cannot necessarily all be derived from a single (formal) source of truth, and therefore we need to envisage and engineer specifications as ecosystems of connected artefacts.

Throughout the paper I discuss a number of observations which have helped us shape the team's philosophy for engineering specifications:

1. Behavioural questions are a unit of specification work.
2. Specifications need to provide accessible, unambiguous, consensual, reproducible and auditable answers.
3. Prose is a necessary but arduous specification medium.
4. Formal and executable specifications can help, but cannot replace prose.
5. The necessity for consensual, accessible prose influences the formalisation choices.
6. Creating programmatic bridges from formalisation to prose seems feasible in practice.
7. Disagreement between specification artefacts is a useful review signal.
8. Certain specifications may need to be engineered as ecosystems of connected artefacts, not necessarily all generated from one single source of truth.
9. Interfaces to query the prose as it is, not as we wish it to be, make prose specifications more directly inspectable.

2 Specifications let us answer behavioural questions

Arm publishes Architecture specifications (the Architecture, for short), which prescribe, for a program in Arm assembly, which behaviours are Allowed or Forbidden. This should be true on all Arm systems: the Architecture describes the envelope of all valid hardware implementations. Thus the Architecture is a contract between hardware and software.

More prosaically, the Architecture allows readers to find answers to questions about the behaviour of Arm systems (e.g. telephones, tablets, laptops, servers...).

An example of a question about the behaviour of an Arm system is: "What does an `LDR` instruction do?" The Architecture lets a user answer that question, as illustrated in Figure 1.

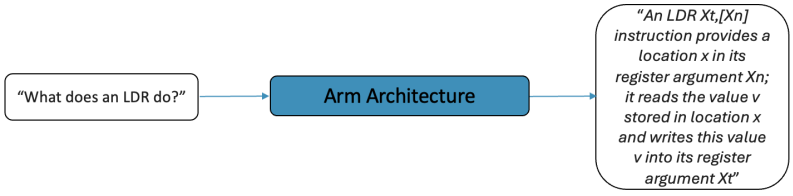


Figure 1 "What does an `LDR` do?"

Another 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. Figure 2 gives two litmus tests, called MP and MP+DMB.ST+DMB.LD.

```
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)
```

```
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 Two examples of questions about the behaviour of an Arm system.

The MP test asks what happens if two threads P0 and P1 communicate via shared memory locations `x` and `y`, as follows:

- 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 MP+DMB.ST+DMB.LD litmus test asks the same question of a program where P0 and P1 have special instructions, called barriers: does this change the outcome?

Figure 3 again illustrates the fact that the Architecture lets the user answer such questions.

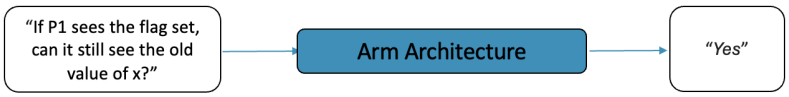


Figure 3 "If P1 sees the flag set, can it still see the old value of `x`?"

Thus, as trivial as it may sound, a specification lets users find out answers to their questions. This leads to the following observation:

Behavioural questions are a unit of specification work.

Ideally, a specification would let users find answers to their questions relatively easily. What easily means may depend on the audience: some will prefer to get answers at the click of a button, others will prefer an answer in high-level prose that they feel to be more accessible, others still will prefer an answer in very precise formal language. At the same time, the ease of finding an answer should not be at the cost of that answer being ambiguous, up for debate, not reproducible, unjustified or wrong. This leads to the following observation:

Specifications need to provide
accessible, unambiguous, consensual, reproducible and auditable answers.

Of course, there are natural tensions between all these traits.

3 Accessibility pulls us towards prose

For accessibility reasons, specifications may be given in prose, as prose may feel a more accessible medium than others. But a specification also needs to provide the users with all the necessary details exhaustively.

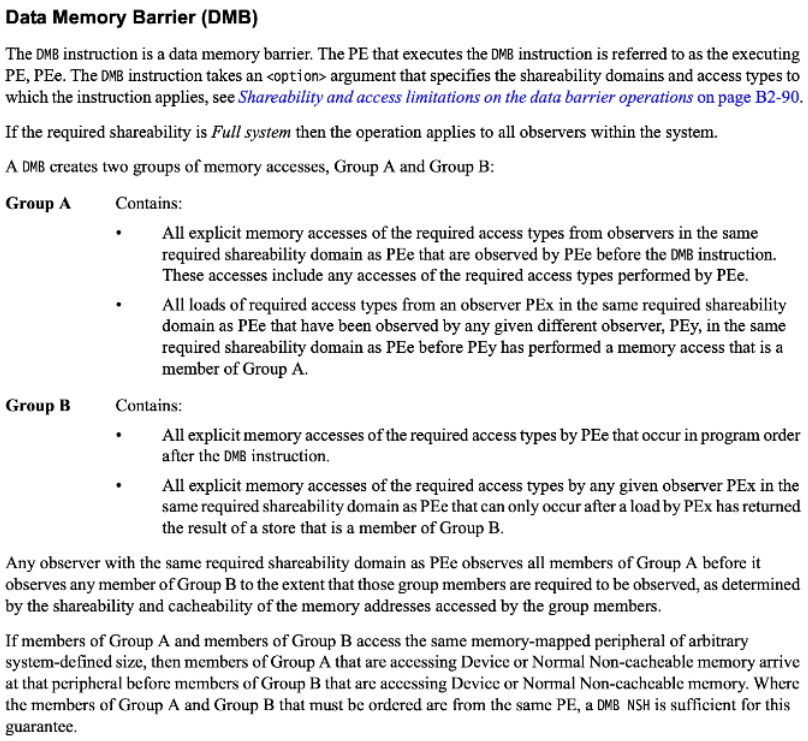
As an illustration, in the context of Arm Architecture, the specification is determined by the Arm Architecture Reference Manual (Arm ARM for short) [8], a technical document made of different types of data, predominantly English prose. The Arm ARM contains the necessary information to answer questions about the behaviour of an Arm system, such as the `LDR` semantics question in Figure 1 and the final state of the MP litmus test in Figure 3.

Whilst the Arm ARM gives a complete specification, which is of course necessary for its suitability as a specification, this affects the length of the document, and therefore its accessibility. The latest issue of the Arm ARM is M.b, which counts 17153 pages: it may be difficult to find the exact elements of response to a given question.

Another example of tensions between desirable traits for specifications is as follows. A prose specification, perhaps therefore perceived as accessible, may however be ambiguous, and therefore invite debates as to how to interpret the specification. Further to this, a prose specification may invite divergences in interpretation, either due to the ambiguity of the

text, or the difficulty in grasping its content. In turn this means that two different readers, or perhaps the same reader several months apart, may reach different answers to a given question, therefore contradicting the notion that answers should be reproducible.

As an illustration, the Arm ARM describes the envelope of legitimate behaviours of relatively complex systems, viz, Arm multiprocessor chips. This means that the specification itself may be complex in places, due to the complexity of the systems it specifies. For example the snapshot of the Arm ARM issue A.k dated 30th of September 2016 given in Figure 4 shows the definition of the barrier instruction DMB at that time. From that, the reader needed to be able to determine whether the behaviour MP+DMB.ST+DMB.LD was Forbidden.



Data Memory Barrier (DMB)

The DMB instruction is a data memory barrier. The PE that executes the DMB instruction is referred to as the executing PE, PEe. The DMB instruction takes an <option> argument that specifies the shareability domains and access types to which the instruction applies, see *Shareability and access limitations on the data barrier operations* on page B2-90.

If the required shareability is *Full system* then the operation applies to all observers within the system.

A DMB creates two groups of memory accesses, Group A and Group B:

Group A Contains:

- All explicit memory accesses of the required access types from observers in the same required shareability domain as PEe that are observed by PEe before the DMB instruction. These accesses include any accesses of the required access types performed by PEe.
- All loads of required access types from an observer PEx in the same required shareability domain as PEe that have been observed by any given different observer, PEy, in the same required shareability domain as PEe before PEy has performed a memory access that is a member of Group A.

Group B Contains:

- All explicit memory accesses of the required access types by PEe that occur in program order after the DMB instruction.
- All explicit memory accesses of the required access types by any given observer PEx in the same required shareability domain as PEe that can only occur after a load by PEx has returned the result of a store that is a member of Group B.

Any observer with the same required shareability domain as PEe observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the shareability and cacheability of the memory addresses accessed by the group members.

If members of Group A and members of Group B access the same memory-mapped peripheral of arbitrary system-defined size, then members of Group A that are accessing Device or Normal Non-cacheable memory arrive at that peripheral before members of Group B that are accessing Device or Normal Non-cacheable memory. Where the members of Group A and Group B that must be ordered are from the same PE, a DMB NSH is sufficient for this guarantee.

Figure 4 A snapshot of the Arm ARM issue A.k dated 30th of September 2016.

This leads to the following observation:

Prose is a necessary but arduous specification medium.

Prose gives users a way into the material, but for large behavioural questions it can ask too much of the reader.

For the longest time, I used to think of the unconstrained English prose that constitutes the vast majority of the information in the Arm ARM as a burden, something that we needed to formalise away to remove ambiguity and simply would not see the end of before we were all very old indeed. I also used to think, rather naively, that once we had written formal models of the Architecture our job was done.

But the contact with my colleagues, engineers and partners, made me rethink that very deeply. Evidently (although it took me years to realise this), we cannot just write a formal model and throw it over the fence, hoping for customers to "just use it", even when we provide tools to execute it.

This leads to the following observation:

Formal and executable specifications can help, but cannot replace prose.

4 Unambiguity and reproducibility pull us towards formalisation and executability

A natural way to provide unambiguous foundations is through formal definitions. For the rest, I once heard Simon Peyton Jones say:

Computers are remorseless but non-judgmental critics
of woolly thinking and faulty logic.

Taken in tandem, these insights point to building formal and executable specifications:

- formal, so that each concept in the specification is precise, well defined;
- executable, so that users can interact with the specification to check their understanding, which in turn can help support consensual foundations and reproducibility of answers.

As an illustration, the Arm Architecture Formal Team develops a number of formal and executable artefacts, which together aim to give a formal and executable specification of the Architecture. At a high level, the Architecture combines instruction semantics for the entirety of the Instruction Set, meaning the Effects that instructions may have on registers and memory, and the Concurrency Model, meaning how those Effects are ordered with respect to one another.

4.1 Concurrency Model

Historically, the concurrency aspects of the Architecture were described in informal prose, as shown in Figure 4. Nowadays, Arm uses a domain-specific language called cat to describe concurrent behaviours. The cat language is implemented in the herd tool [5, 6], which lets users ask behavioural questions phrased as litmus tests, and obtain reproducible answers.

The Arm Architecture Formal Team develops, maintains and enhances the cat file and the associated herd, diy [3] and litmus [4] tools and regularly upstreams their contributions to GitHub [6]. The website `developer.arm.com/herd7` hosts an interface to the herd tool.

At a high-level, the Arm Concurrency Model gives ordering requirements over memory accesses, stating which local orderings must be respected by hardware. Those constraints forbid certain shapes in program executions. The External visibility requirement forbids cycles in the Ordered-before relation, which consists of observations between threads (Observed-by relation) and local orderings (Hardware-Required-Ordered-Before relation) [8, Chapter B2.3].

For example, let's examine the Forbidden execution of MP+DMB.ST+DMB.LD given in Figure 5. Each pink box gives the semantics of one of the two threads P0 and P1.

On each thread, the semantics of each instruction is given in terms of Effects; for example:

- The store instruction STR W0,[X1] on P0 corresponds to an Explicit Memory Write Effect a:W[x]=1.
- The STR W2,[X3] corresponds to b:W[y]=1.
- The load instruction LDR W4,[X3] on P1 corresponds to an Explicit Memory Read Effect c:R[y]=1.
- The LDR W5,[X1] corresponds to d:R[x]=0.
- The DMB ST barrier instruction on P0 corresponds to a Barrier Effect j:DMB.ST.

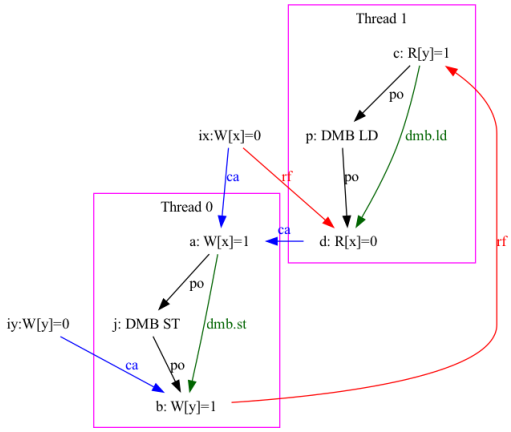


Figure 5 A Forbidden execution of MP+DMB.ST+DMB.LD.

Effects are connected by relations of various kinds, as shown in Figure 5.

The main point here is not the particular names of the relations. The Concurrency Model computes relations over the candidate execution and rejects it because these relations create an Ordered-before cycle. Hence a behavioural question, phrased as a litmus test, is answered unambiguously and in a reproducible manner.

4.2 Instruction Semantics

When we looked at the Forbidden execution of MP+DMB.ST+DMB.LD, we mapped a store instruction STR to a Memory Write Effect, a load instruction LDR to a Memory Read Effect, and a barrier instruction to a Barrier Effect. We implicitly or unconsciously assumed that x and y were physical memory locations. But usually one interacts with virtual memory addresses rather than physical memory directly.

The mapping from virtual addresses to physical locations resides in a collection of *page tables*, or *translation tables* in Arm jargon. In other words, translation tables are used to translate the virtual addresses seen by software into physical locations used by hardware.

Thus the semantics of instructions need to adjust to this level of abstraction: for example, a load instruction LDR does not simply map to an Explicit Memory Read Effect like it did at user-level abstraction. Instead, the semantics of LDR at the virtual memory level of abstraction needs to account for accesses made to translation tables, which determine which physical memory location the LDR can read from, if any.

Consider e.g. the left-hand-side of Figure 6: the semantics of the LDR instruction is contained within the pink box. There the load starts off with an Implicit Memory Effect a:R[TTD(x)]=(oa:PA(x)), which reads the translation table entry for the virtual address x, and gets a physical address PA(x). This lets us know where in memory the load should be taking its value from, which is represented by the Explicit Memory Effect b:R[PA(x)]=1.

Each translation table descriptor (TTD) indicates whether the entry is valid or not:

- If it is valid, reading the TTD gives the physical location corresponding to a virtual address as is the case on the left-hand-side of Figure 6.
- If it is not valid, accessing a virtual address via that TTD raises an Exception: the right-hand-side of Figure 6 shows the case where the TTD of the virtual address x was invalid, leading to Exception Entry Effect g.

Historically, we write these instruction semantics by hand into the source of herd. We invent these semantics through educated guesses after reading the Arm ARM, reviewing our candidate semantics with colleagues and partners, and testing them against hardware.

But the Arm ARM also describes each instruction in the Arm Instruction Set in terms of a piece of code in a language called Architecture Specification Language (ASL for short). Figure 7 shows an excerpt of ASL code for the LDR instruction [11].

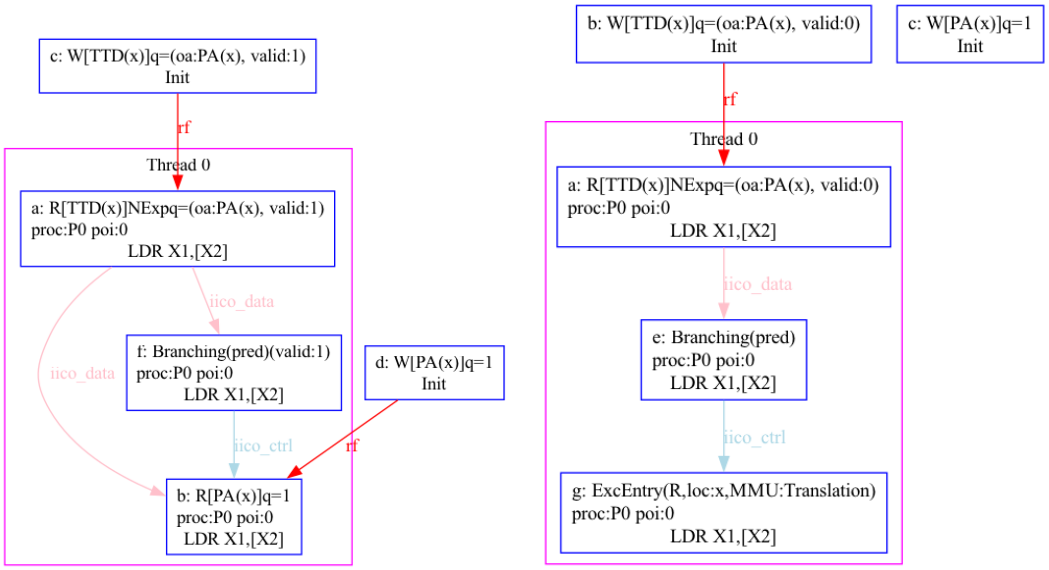


Figure 6 Aspects of the virtual memory semantics of LDR .

Operation

```
var address : bits(64);

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                  tagchecked, t);

if n == 31 then
    CheckSPAlignment();
    address = SP{64}();
else
    address = X{64}(n);
end;

if !postindex then
    address = AddressAdd(address, offset, accdesc);
end;

let data : bits(datasize) = Mem{datasize}(address, accdesc);
X{regsize}(t) = ZeroExtend{regsize}(data);

if wback then
    if wb_unknown then
        address = ARBITRARY : bits(64);
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    end;
    if n == 31 then
        SP{64}() = address;
    else
        X{64}(n) = address;
    end;
end;
```

Figure 7 Excerpt of ASL code for the LDR instruction.

Historically, ASL did not have any defined syntax nor semantics, although implementations existed, e.g. [16, 17]. Arm decided to invest in a formal definition of that language, and make some adjustments to the syntax along the way. The resulting language is called ASL1, whilst

the old pseudocode language is referred to as ASL0. The Arm ARM has now entirely moved to be in ASL1, since issue M.b. The ASL Reference documentation [9], tooling (type-checker and interpreter) [12] and standard library proofs [10] are developed, maintained and enhanced by the Architecture Formal Team. Arm also provides, as for herd, a web interface to the ASLRef tooling [14].

Work is well under way to be able to link, from a formal and executable point of view, the instruction semantics given by the ASL code of instructions, and the handwritten semantics present in herd [18]. This should allow much more scalability in handling instructions in herd, and expand the scope of what the formal Arm Concurrency Model covers.

Naturally our work on ASL is adjacent to the work on Sail [15]. Arm chose not to use Sail for a number of reasons:

- the syntax of Sail was considered to be unintuitive by Architects;
- we could not seem to find a formal semantics of Sail.

5 Consensual prose shapes formalisation

The readership of the Arm ARM is very large – potentially the entire Arm ecosystem – and not necessarily equipped with formal knowledge.

Thus a rather large amount of the Architecture Formal Team's time is spent translating our formal definitions and findings into hopefully relatively accessible insights. We do this socially with our stakeholders, by composing emails, hosting meetings and writing slides.

A relatively simple illustration of the consequences of our social situation – the necessity to communicate our artefacts in ways that are intelligible and accessible to our audience – is as follows. The Arm Concurrency Model defines a relation called Ordered-before, ob in cat, which is the transitive closure of the union of two other relations as discussed above, viz, Hardware-Required-Ordered-Before (hw-reqs in cat) and Observed-by (obs in cat). Hence the cat definition of Ordered-before could (but is not) phrased very succinctly thus:

```
let ob = (hw-reqs | obs)+
```

The transliteration of this cat clause, in other words the English prose that would appear in the Arm ARM and that most readers would refer to instead of the cat, would necessarily have to refer to the notion of the transitive closure. This is because we aim to have the cat and the transliterated English be as close to each other as possible, in particular for debuggability reasons.

But the notion of transitive closure may not be entirely natural or known by our audience, and therefore we needed to find a different cat phrasing for Ordered-before, a phrasing whose transliteration would be more readily accessible to our audience, whilst remaining semantically equivalent to the transitive closure. As a consequence, the definition of Ordered-before in cat is thus:

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

This is then transliterated to the following [8, Section B2.3.9], where the transitive closure is phrased explicitly:

> *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.*

This leads to the following observation:

The necessity for consensual, accessible prose influences the formalisation choices.

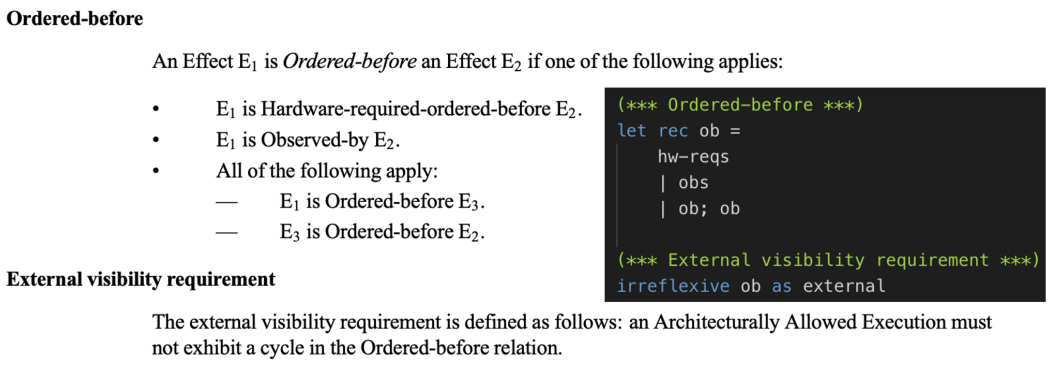
6 Engineering specifications as ecosystems of connected artefacts

Developing these formal definitions is a slow and partially social process. For example, for the Concurrency Model:

- We read the prose in the Arm ARM.
- We test machines.
- We interview our colleagues (Architects, implementors, software developers): we show them hundreds to thousands of tests and ask them if they think the behaviours should be Allowed or Forbidden.
- We extrapolate a set of definitions, which we regress against deployed hardware;
- We ratify the definitions with Arm partners.
- Then the transliteration of these definitions lands in the Arm ARM.

Indeed, the Architecture is determined by what is given in the Arm ARM. In other words, if the cat file does not appear somehow in the Arm ARM, it is not an authoritative reference point. Thus there is a need to transliterate the cat file into the Arm ARM.

By convention, we aim for 1:1 correspondence between cat and English transliteration, to help us "debug" the prose. Indeed we can execute and test the cat file, and the immediacy of the correspondence between the cat file and its English transliteration helps convey some of that debuggability to the prose. Figure 8 shows a fragment of the cat file and its corresponding English transliteration in the latest issue M.b of the Arm ARM.



Ordered-before

An Effect E_1 is *Ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Hardware-required-ordered-before E_2.
- E_1 is Observed-by E_2.
- All of the following apply:
  - E_1 is Ordered-before E_3.
  - E_3 is Ordered-before E_2.

```
(*** Ordered-before ***)
let rec ob =
    hw-reqs
    | obs
    | ob; ob

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

External visibility requirement

The external visibility requirement is defined as follows: an Architecturally Allowed Execution must not exhibit a cycle in the Ordered-before relation.

Figure 8 Ordered-before fragment of the latest issue M.b of the Arm ARM and the corresponding fragment of the cat file.

Since this formalisation effort started, we added quite a few features, such as virtual memory and address translation [2].

With all the added features and extensions, the cat file and the corresponding definitions have grown tremendously. Figure 9 shows the definition of Hardware-Required-Ordered-Before in the latest issue M.b of the Arm ARM.

Evidently, the more the cat file grows, the more the prose grows, and the transliteration becomes a rather arduous exercise. Since the Arm ARM is the reference definition for the whole Arm ecosystem, what seems like a modest typo at our end can have consequences on hardware designs, or software developments.

Locally-ordered-before

An Effect E_1 is *Locally-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Tag-check-intrinsically-before E_2.
- E_1 is Translation-intrinsically-before E_2.
- E_1 is Fetch-intrinsically-before E_2.
- E_1 is ETS-ordered-before E_2.
- E_1 is Fetch-ordered-before E_2.
- E_1 is Same-Cache-Line-ordered-before E_2.
- E_1 is DSB-ordered-before E_2.
- E_1 is Instruction-fetch-barrier-ordered-before E_2.
- E_2 is a Local memory write successor of E_1.
- All of the following apply:
  - E_3 is a Local memory write successor of E_1.
  - E_3 belongs to the same single-copy-atomic grouping as E_2.
- E_1 is Dependency-ordered-before E_2.
- E_1 is Pick-ordered-before E_2.
- E_1 is Atomic-ordered-before E_2.
- E_1 is Barrier-ordered-before E_2.
- All of the following apply:
  - E_1 is Locally-ordered-before E_3.
  - E_3 is Locally-ordered-before E_2.

Pick-locally-ordered-before

An Effect E_1 is *Pick-locally-ordered-before* an Effect E_2 if all of the following apply:

- There is a Pick dependency from E_1 to E_3.
- E_3 is Locally-ordered-before E_2.
- E_2 is an Explicit Memory Write Effect.

Locally-hardware-required-ordered-before

An Effect E_1 is *Locally-hardware-required-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Locally-ordered-before E_2.
- E_1 is Pick-locally-ordered-before E_2.
- All of the following apply:
  - E_1 is Locally-hardware-required-ordered-before E_3.
  - E_3 is Locally-hardware-required-ordered-before E_2.

Hardware-required-ordered-before

An Effect E_1 is *Hardware-required-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Locally-hardware-required-ordered-before E_2.
- E_1 is Hazard-ordered-before E_2.

Figure 9 Definition of Hardware-Required-Ordered-Before in issue M.b of the Arm ARM.

So we invented a tool called miaou, which takes the cat file and produces the constrained English prose of the Arm ARM programmatically. This tool is distributed on GitHub [6], and we have used it since the Arm ARM issue K.a (dated 20 March 2024).

This leads to the following observation:

Creating programmatic bridges from formalisation to prose seems feasible in practice.

The insight here is not to build a prose generator, but rather this idea of engineering a bridge, to keep a formal and executable artefact and authoritative prose close enough that the prose can be debugged by proxy.

This idea of generating prose from the formal and executable artefact is reminiscent of the approach taken by the WebAssembly ecosystem with SpecTec [19]: having one single source of truth out of which other representations are derived.

Whilst this is a tempting approach, in the context of our work we need to depart from this ambition a little. Indeed we are not, and cannot be, the only authors of the specification: different parts of the Arm ARM are authored by separate teams with different methodologies, tools or workflows. The vast majority of the material that appears in the Arm ARM is relatively unconstrained prose written by Arm Architects to describe the rules of their respective areas of the Arm Architecture. Figure 10 gives an illustration of prose written by Arm Architects, an excerpt of the chapter on Address Translation.

The lesson is not that single-source approaches are undesirable. They are attractive when one authoritative authoring process can produce all the representations users need. In our setting, however, different communities author different parts of the Architecture, different users need different representations, and the authoritative prose cannot simply be replaced by a formal artefact.

D8.1 Address translation

R_{DWFWP} If an implementation is executing in AArch64 state, then that implementation uses either or both of the VMSAv8-64 and the VMSAv9-128 translation systems.

R_{RKHJV} All of the following determine whether a translation stage uses the VMSAv9-128 translation system:

- For stage 1 translations in the EL1&0 translation regime, the Effective value of TCR2_EL1.D128 is 1.
- For stage 1 translations in the EL2&0 translation regime, the Effective value of TCR2_EL2.D128 is 1.
- For stage 1 translations in the EL3 translation regime, the Effective value of TCR_EL3.D128 is 1.
- For stage 2 translations, the Effective value of VTCR_EL2.D128 is 1.

Otherwise, the translation stage uses the VMSAv8-64 translation system.
This applies even when the translation stage is disabled.

I_{CCTQS} Address translation converts the addresses used by instructions to the addresses used by the physical memory system.

R_{CRPHV} When a data address or instruction address is used in an instruction, it is a *virtual address* (VA). This includes any address stored in one of the following registers:

- *Program counter* (PC).
- *Stack pointers* (SP).
- *Link register* (LR).
- *Exception link register* (ELR).

R_{CRDGS} When an access is made to the physical memory system, a *physical address* (PA) is used.

R_{KCNRX} An address translation maps a VA to a PA.

R_{CCCQQ} An address translation requires one of the following:

- A single translation stage, stage 1.
- Two sequential translation stages, stage 1 and stage 2.

R_{ZKJWW} An address translation stage maps an *input address* (IA) to an *output address* (OA).

R_{BCKLH} If one address translation stage is used, then a VA is mapped to a PA using all of the following steps:

1. The VA is input as the IA to the translation stage.
2. The PA is output as the OA from the translation stage.

Figure 10 English prose written by Arm Architects from Arm ARM M.b .

Thus the Arm ARM features a number of different types of Architectural Data, including:

1. The ASL code for each instruction – see e.g. Figure 7.
2. The constrained English prose programmatically generated from the cat file using the miaou tool – see e.g. Figure 9.
3. The less constrained English prose written by Arm Architects to describe the rules of their respective areas of the Arm Architecture – see e.g. Figure 10.

Specification engineering often requires manipulating the Architectural Data in a number of ways, for example:

- Producing Architectural Data when it does not exist (e.g. the Concurrency Model).
- Giving meaning to Architectural Data (e.g. the ASL1 semantics).
- Interpreting Architectural Data to answer questions.
- Preparing Architectural Data to make it suitable for analysis.
- Confronting two sources of Architectural Data.

This is also why multiple artefacts are useful. Hardware experiments can confront a model with observed behaviour. A model can confront prose with its formal consequences. Generated prose can confront a formal definition with the demands of readability. A query interface can confront the manual with the questions users actually ask. Expert review can confront all of these with architectural intent.

Disagreement is one of the ways the socio-technical system that develops, maintains and uses the Architecture learns and improves. Of course, unresolved disagreement in a published specification is a problem. But during development, tension between artefacts is often the signal that something needs to be understood more clearly.

This leads to the following observation:

Disagreement between specification artefacts is a useful review signal.

Instruction semantics is a concrete example of this tension. Aspects of an instruction such as `LDR` appear as Effects consumed by the Concurrency Model (see Figure 6), as ASL code (see Figure 7) and as prose (e.g. Figure 10). These representations serve different audiences and different tools.

Hence specifications can quickly become ecosystems of connected representations. For us, this points to trying to engineer specifications as a collection of connected artefacts, rather than one single artefact trying to meet all these somewhat contradicting traits.

This leads to the following observation:

Certain specifications may need to be engineered as ecosystems of connected artefacts, not necessarily all generated from one single source of truth.

7 Working to make prose specifications more directly inspectable

Up until recently, we only exploited the less constrained English prose written by Architects through reading and conceptualising.

More recently, LLMs took the world by storm, and allowed us to interrogate large bodies of prose, so we started experimenting with them:

- could they help us examine the unconstrained prose more immediately, not by proxy as we do for the Concurrency Model?
- could they help us make our formal artefacts more accessible, acting as informal interfaces to the formal tools?

An initial result of our experiments is a grounded query interface called The Architecture Speaks, that is hosted on the Arm website [13, 1]. This interface is experimental, and a work in progress, but can already answer behavioural questions such as the ones we discussed in the introduction, as shown in Figure 11, and accessible at [7].

We have taken great care in making sure that the query interface points the user back to the Arm ARM location of the material used in the answer, thereby making its answers grounded in the source material. As one can see, we seem to be miles away from formal specifications all of a sudden. But we are already finding that people find this tool to be useful, simply to navigate the Arm ARM.

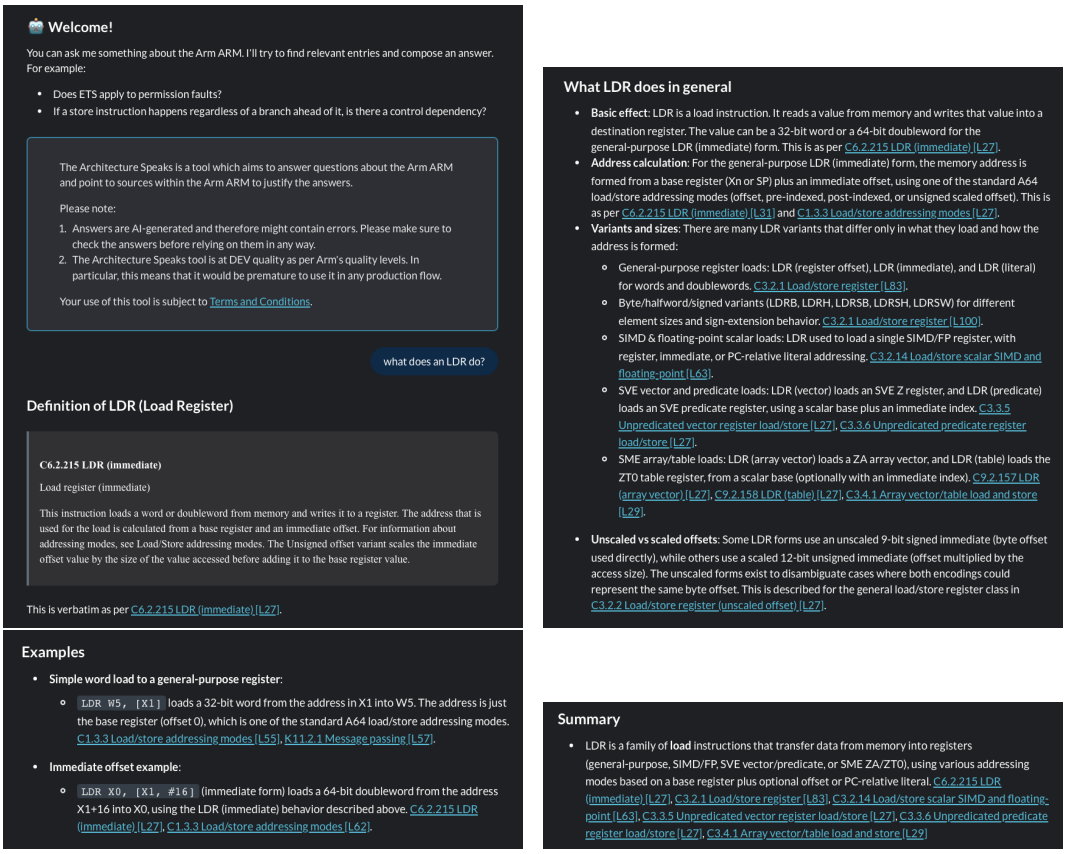


Figure 11 Snapshot from The Architecture Speaks on the question "What does an LDR do?" [7].

The main point is not so much the relative novelty of a query interface for the Arm ARM. Instead, the point is auditability of the current artefact, as it is written. A grounded query interface lets us ask whether the manual, as it is written, supports the questions users actually ask. This leads to the following observation:

Interfaces to query the prose as it is, not as we wish it to be,
make prose specifications more directly inspectable.

The interface is not meant to be an oracle, nor to replace the Arm ARM. Its purpose is to make the current prose more inspectable. This is why answers must be grounded in source locations, why users must be able to audit the material used in the answer, and why failures to answer are themselves useful signals: they may indicate that the prose does not support a user question clearly enough.

8 Towards computer-aided architecture development and consumption

The Arm Architecture Formal Team's experience suggests elements of a philosophy of specification engineering that may be reusable in other contexts:

1. Treat behavioural questions as units of specification work.
2. Decide what kind of answer each behavioural question needs: accessible, unambiguous, consensual, reproducible, auditable, maintainable, layerable.
3. Accept that perhaps a single artefact may not satisfy all those properties.

4. Engineer explicit bridges between artefacts.
5. Use disagreement between artefacts as a review signal.
6. Inspect the prose users actually read.

Figure 12 shows how these elements arise from the examples in this paper.

Question	Artefacts	Properties
What does an `LDR` do?	Arm ARM prose, ASL, query interface	Accessibility, auditability
Is MP Allowed?	Litmus test, cat model, herd	Unambiguity, reproducibility
Why is the barrier MP test Forbidden?	Execution graph, Ordered-before relation, cat model	Unambiguity, auditability
How to write Ordered-before?	cat definition, Arm ARM transliteration, stakeholder review	Consensuality, accessibility, auditability

Figure 12 Behavioural questions as units of specification work.

We believe that, as shown in Figure 13, certain specifications may need to be maintained as connected artefact ecosystems, with prose, formal models, generated bridges, executable tools, expert review and grounded query interfaces each contributing different kinds of answer.

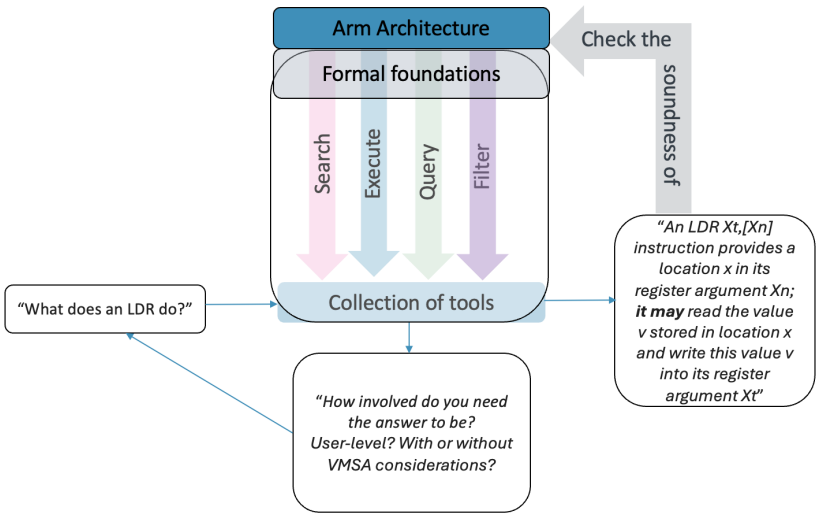


Figure 13 Vision for the Arm Architecture Formal Team.

References

1 Jade Alglave. Introducing "The Architecture Speaks", 2026. URL: `https://developer.arm.com/community/arm-community-blogs/b/ai-blog/posts/introducing-the-architecture-speaks`.

2 Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, and Nikos Nikoleris. Puss In Boots: On Formalizing Arm's Virtual Memory System Architecture. *IEEE MICRO*, 2024. `doi:10.1109/MM.2024.3422668`.

3 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in Weak Memory Models. *CAV*, 2010.

4 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. *TACAS*, 2011.

5 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats. *TOPLAS*, 2014.

6 Jade Alglave and Luc et al Maranget. herdtools7, 2026. URL: `https://github.com/herd/herdtools7`.

7 The Architecture Speaks chatbot. "what does an LDR do?", 2026. URL: `https://developer.arm.com/architecture/the-architecture-speaks/?threadId=228d9997-7d58-4b2d-a1ce-2bfd6339aec3`.

8 Arm. Arm Architecture Reference Manual for A-Profile Architecture. URL: `https://developer.arm.com/documentation/ddi0487/mb/?lang=en`.

9 Arm. ASL Reference, 2025. URL: `https://developer.arm.com/architectures/architecture%20specification%20language`.

10 Arm. ASL stdlib proofs, 2025. URL: `https://github.com/herd/herdtools7/tree/master/acl2/asl/proofs/stdlib`.

11 Arm. ASL1 code for LDR, 2025. URL: `https://developer.arm.com/documentation/ddi0602/2025-12/Base-Instructions/LDR--immediate---Load-register--immediate--`.

12 Arm. ASLRef tooling, 2025. URL: `https://github.com/herd/herdtools7/tree/master/asllib`.

13 Arm. The Architecture Speaks, 2026. URL: `https://developer.arm.com/architecture/the-architecture-speaks/`.

14 Arm. Web interface to ASLRef tooling, 2026. URL: `https://developer.arm.com/architectures/tools/aslref`.

15 Kathryn E. Gray, Peter Sewell, Christopher Pulte, Shaked Flur, and Robert Norton-Wright. The Sail instruction-set semantics specification language. URL: `https://www.cl.cam.ac.uk/~pes20/sail/manual.pdf`.

16 Alastair Reid. Trustworthy specifications of Arm v8-A and v8-M system level architecture. *FMCAD*, 2016.

17 Alastair Reid. ASL0 interpreter, 2020. URL: `https://github.com/alastairreid/asl-interpreter`.

18 Hadrien Renaud. Executable semantics of Arm's Architecture Specification Language. *JFLA*, 2024. URL: `https://hal.science/hal-04406399/document`.

19 Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. Bringing the WebAssembly Standard up to Speed with SpecTec. URL: `https://dl.acm.org/doi/10.1145/3656440`.