ArchSem: Reusable Rigorous Semantics of Relaxed Architectures

THIBAUT PÉRAMI, University of Cambridge, United Kingdom
THOMAS BAUEREISS, University of Cambridge, United Kingdom
BRIAN CAMPBELL, University of Edinburgh, United Kingdom
ZONGYUAN LIU, Aarhus University, Denmark
NILS LAUERMANN, University of Cambridge, United Kingdom
ALASDAIR ARMSTRONG, University of Cambridge, United Kingdom
PETER SEWELL, University of Cambridge, United Kingdom

The specifications of mainstream processor architectures, such as Arm, x86, and RISC-V, underlie modern computing, as the targets of compilers, operating systems, and hypervisors. However, despite extensive research and tooling for instruction-set architecture (ISA) and relaxed-memory semantics, recently including systems features, there still do not exist integrated mathematical models that suffice for foundational formal verification, of concurrent architecture properties or of systems software. Previous proof-assistant work has had to substantially simplify the ISA semantics, the concurrency model, or both.

We present ArchSem, an architecture-generic framework for architecture semantics, modularly combining ISA and concurrency models along a tractable interface of instruction-semantics effects, that covers a range of systems aspects. To do so, one has to handle many issues that were previously unclear, about the architectures themselves, the interface, the proper definition of reusable models, and the Rocq and Isabelle idioms required to make it usable. We instantiate it to the Arm-A and RISC-V instruction-set architectures and multiple concurrency models.

We demonstrate usability for proof, despite the scale, by establishing that the Arm architecture (in a particular configuration) provides a provable virtual memory abstraction, with a combination of Rocq, Isabelle, and paper proof. Previous work provides further confirmation of usability: the AxSL program logic for Arm relaxed concurrency was proved sound above an earlier version of ArchSem.

This establishes a basis for future proofs of architecture properties and systems software, above production architecture specifications.

CCS Concepts: • **Computer systems organization → Architectures**; • **Theory of computation → Semantics and reasoning**.

Additional Key Words and Phrases: Semantics, Architectures, Theorem Provers

ACM Reference Format:
Thibaut Pérami, Thomas Bauereiss, Brian Campbell, Zongyuan Liu, Nils Lauermann, Alasdair Armstrong, and Peter Sewell. 2026. ArchSem: Reusable Rigorous Semantics of Relaxed Architectures. *Proc. ACM Program. Lang.* 10, POPL, Article 8 (January 2026), 31 pages. https://doi.org/10.1145/3776650

Authors' Contact Information: Thibaut Pérami, University of Cambridge, Cambridge, United Kingdom, thibaut.perami@cl.cam.ac.uk; Thomas Bauereiss, University of Cambridge, Cambridge, United Kingdom, Thomas.Bauereiss@cl.cam.ac.uk; Brian Campbell, University of Edinburgh, Edinburgh, United Kingdom, Brian.Campbell@ed.ac.uk; Zongyuan Liu, Aarhus University, Aarhus, Denmark, zy.liu@cs.au.dk; Nils Lauermann, University of Cambridge, Cambridge, United Kingdom, Nils.Lauermann@cl.cam.ac.uk; Alasdair Armstrong, University of Cambridge, Cambridge, United Kingdom, Alasdair.Armstrong@cl.cam.ac.uk; Peter Sewell, University of Cambridge, Cambridge, United Kingdom, Peter.Sewell@cl.cam.ac.uk.

ACM 2475-1421/2026/1-ART8
https://doi.org/10.1145/3776650

1 Introduction

The problem we address in this paper is the fact that there still does not exist a generally usable rigorous semantics for the behaviour of any modern relaxed CPU architecture – one that suffices for foundational formal proofs, both about the architecture as a whole and about specific human-written or compiler-generated code. In principle, an architecture specification defines the abstraction that hardware provides to software, abstracting from the microarchitectural details of hardware implementation. To actually define the behaviour allowed by an architecture with mathematical precision – the range of permitted behaviour of an arbitrary initial machine state on an arbitrary hardware implementation – one needs two main components:

- the behaviour of single instructions in isolation, the *Instruction Set Architecture* (ISA) semantics; and
- the *relaxed concurrency model* that determines which writes each read (and each instruction-fetch read, and each translation-walk read) can read from in an allowed execution.

The last 15 years have seen much progress in both. On the ISA side, Arm transitioned from definitions in pseudocode to a mechanised (but not formalised) ASL language by Reid [53], and full-scale definitions of the Arm A-profile architecture (henceforth just 'Arm') and the Arm CHERI Morello architecture have been automatically translated from ASL to Sail, and thence to provers for formal reasoning [14, 15, 17]. RISC-V International develop an official Sail ISA model [44], as do the CHERI RISC-V and CHERIoT design teams [11, 68]. For x86, there are substantial third-party models by Goel et al. [30], in ACL2 and also translated into Sail [19], Dasgupta et al. [23], and Roessle et al. [56]; and there is work in Intel towards a vendor reference model by Reid [54].

On the concurrency side, extensive research has put 'user' relaxed concurrency for x86, Arm, RISC-V, and IBM Power on a solid footing, e.g. [1, 2, 4, 8–10, 16, 21, 25, 27–29, 31, 35, 49, 50, 52, 58–61, 65], but that did not touch on 'systems' relaxed concurrency. Recent work by Simner et al. [62, 63, 64] and by Alglave et al. [6][13] developed semantics for many aspects of Arm instruction fetch and instruction-cache maintenance, virtual memory, and exceptions. These models have been developed and validated with a combination of experimental testing and discussions with Arm architects. While doubtless still subject to change, they finally provide a reasonably solid foundation for these aspects of concurrent systems code.

Much of this latter work has emphasised making the concurrency models executable as test-oracle software tools that can compute the allowed behaviour of litmus tests, from memevents [60] onwards and including RMEM [24], Herd [10], and Isla-axiomatic [16]. Each of these involves some integration of ISA and concurrency in the tooling, variously for fragments or complete ISAs – but not as mathematical definitions that one can reason about. Instead, each adds implicit semantic content when connecting the concurrency model and ISA semantics, described only in the tool source-code. For example, for Isla-axiomatic and Herd, important aspects of the candidate execution generator are hardcoded within the tools, which can only be discovered by inspecting the highly complex implementation, alongside some paper maths definitions [3]. There have also been some mechanisations of relaxed models, e.g. by Vafeiadis [66] and Podkopaev et al. [47, 48], and mechanised proof of equivalence between promising and axiomatic models by Pulte [49], Pulte et al. [51], but all these are limited in many ways, e.g. without ISA integration and systems features.

It is therefore high time to develop an integrated mathematical semantics for modern architectures. Given the scale, it necessarily has to be embedded in a theorem prover, not paper mathematics, to be useful (the scale also creates prover challenges in itself). We do so largely in Rocq and partly in Isabelle, though much of the work could usefully be ported to other systems too.

We aim for this to be *generic*, supporting multiple architectures, including Arm and RISC-V; multiple styles of relaxed concurrency model, including axiomatic, operational, and promising; and both user and systems aspects. We engineer the definitions to be uniform across architectures where possible and parameterised where necessary. And we aim for it to be *general purpose*, supporting both proof of metatheory and execution for testing. There are many potential uses of such a semantics: for proofs of fundamental properties of an architecture, e.g. that it can provide a virtual-memory abstraction; as foundational semantics for systems software verification, where previous compiler and OS verification projects such as [22, 32, 36–42] have had to ignore or idealise the underlying Arm ISA and/or concurrency behaviour; as a basis for equivalence and refinement proofs between concurrency models; for soundness proofs of higher-level reasoning methods; and as a fully formal reference to cross-check existing concurrency models and tools against.

Our first main contribution is ArchSem, a generic and general-purpose Rocq framework to define formal models of CPU architectures, integrating ISA and relaxed concurrency semantics. We:

(1) Develop an algebraic-effect-based interface in Rocq to modularly connect ISA and concurrency models (§2,3), along with an Isabelle version of the interface (§4.3). This required new Rocq libraries for inductive free monads and computable finite relations.
(2) Define in Rocq a general notion of architecture semantics, generic over axiomatic, operational, and promising styles, and, for axiomatic models, general notions of candidate executions (§5.2). This is set up to support both executable and non-executable models.

Second, we instantiate this in various ways:

(1) With ISA semantics for the complete Armv9.4-A and RISC-V ISAs, and a handwritten Tiny-Arm model (§4). The first two are automatically translated from the authoritative vendor specifications, suitably adapted, to Rocq and Isabelle – respectively Sail automatically translated from the Arm ASL, and the Sail developed by RISC-V International. This translation also generates instantiations of the architecture-specific interface type parameters.
(2) With various relaxed concurrency models: user-mode and virtual-memory Arm axiomatic models (§5.2); a user-mode Arm Promising model (§5.5); a user-mode RISC-V axiomatic model; and sequential operational and sequentially consistent (SC) axiomatic models (§5.4).

Third, we exercise the framework, showing that it is usable for substantial proofs:

(1) Our main theoretical result is a virtual-memory abstraction theorem for Arm (§6). Operating systems use the architecture to provide an abstraction of virtual memory (VM) to user programs: if the architecture's virtual memory behaviour is properly designed, and the OS properly configures it, then arbitrary user code should behave according to the *user-mode* relaxed memory and ISA semantics, and without access to the memory of other processes or the OS. This is a key property about the architecture itself. We prove such a result, modulo some relatively minor assumptions, with a combination of Rocq, Isabelle, and paper proof, for the complete Armv9.4 ISA and the VM axiomatic model of [63].
(2) As a further exercise in use of the framework for proof, we show in Rocq that the sequential operational model refines the SC axiomatic model (§5.4).
(3) Other work, not presented here, provides additional confirmation that the framework is usable: the soundness proof of the AxSL Iris program logic for relaxed user Arm [33] is above an earlier version of the interface, reasoning about syntactic machine instructions in a way that scales with their decomposition into effects (there, "micro-instructions").

In doing all this, we have to handle many things that one might not think of at first sight: about the architectures themselves (e.g. it raised subtle questions about Arm mixed-size semantics, resolved by lengthy discussion with Arm); about the interface between ISA and concurrency; about the proper definition of reusable models (especially partial candidate executions and intra-instruction parallelism); and about the Rocq and Isabelle idioms required to make the whole usable.

This is still far from the ultimate goal of a complete, definitive, and well-validated mathematical semantics for any modern architecture, of course. Indeed, that may not be achievable even in principle, and it certainly is not right now: architecture design continues apace, and many aspects of all the main architectures remain unexplored from a semantic point of view. However, what we describe does provide a sufficient foundation for substantial metatheory.

There are two previously unaddressed issues for which we develop approaches in paper maths that are not currently in the mechanisation:

(1) At present the Rocq implementation is limited to (the large class of) linearly ordered instructions (§3.1); we describe a scheme for the more general intra-instruction concurrency which other instructions require (§7.1).
(2) We develop an approach for concurrency models to support undefined behaviour (§3.3,7.2).

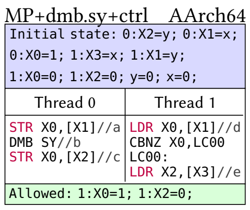
Finally, we highlight the limits of the paper and of the current state of the art, which also provides a roadmap for future research:

(1) For user-level (non-systems) relaxed concurrency, we have instantiated ArchSem for Arm and RISC-V but not (for example) x86 and Power. Given the known user models for the latter, we expect ArchSem to smoothly extend to them.
(2) On the relaxed systems concurrency side, research has so far focussed on Arm: to the best of our knowledge there are no established systems-level concurrency models for RISC-V, x86, or Power. However, in broad terms they are similar, and we expect the framework will support system models for them without substantial change.
(3) Even for Arm, while there are reasonably solid models for many aspects of instruction fetch, virtual memory, and exceptions, there remain open questions within each of these, and their interactions have not yet been thoroughly investigated.
(4) ArchSem is designed from the outset to support execution, both within Rocq and via extraction, but this is not yet well exercised. In particular, there do not yet exist ArchSem-based analogues of model-exploration tools such as rmem, herd, or isla-axiomatic.
(5) We illustrate the ArchSem support of operational models with Arm user sequential and promising, but do not yet have ArchSem versions of operational models such as Flat [26].
(6) ArchSem should support the Arm official axiomatic models, perhaps with minor extensions, but we have not yet tried to automatically import those, which will require analysis of the implicit semantics of herd.
(7) Following most previous relaxed concurrency research, we focus on finite executions.
(8) ArchSem is designed to support both the complete ISA semantics described above, as exported by Sail to Rocq, and the result of symbolically specialising them to particular opcodes under given assumptions, as done by Isla [16] and Islaris [57], but the latter is not yet exercised. Deep embeddings of Sail models, along the lines of [34], may also be useful.

There are other aspects that we leave entirely out of scope at present, to keep this manageable: we do not consider whole-System-on-Chip (SoC) aspects, including the Arm Generic Interrupt Controller and IOMMU; we do not support external I/O (memory mapped devices or x86 ports); and we focus solely on CPU architectures, leaving GPU and other accelerators aside.

The developments are available open-source [18, 46]. An artifact containing source code and paper proofs is available at the Cambridge Apollo data repository [45]. In the text we sometimes simplify for presentation where specified, and we omit Rocq typeclass machinery throughout.

§2 gives context and a simplified overview. §3 defines our interface between ISA and concurrency models. §4 describes how we used Sail to get well-validated ISA models matching our interface, then §5 combines those with axiomatic (§5.2) and promising (§5.5) concurrency models, to define full CPU architecture models. §6 gives our abstraction theorem and §7 discusses future directions.



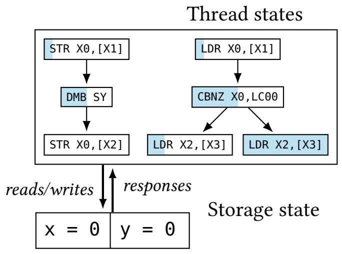
MP+dmb.sy+ctrl AArch64

```
Initial state: 0:X2=y; 0:X1=x;
0:X0=1; 1:X3=x; 1:X1=y;
1:X0=0; 1:X2=0; y=0; x=0;
```

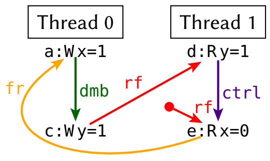
Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//d`
`DMB SY//b`	`CBNZ X0,LC00`
`STR X0,[X2]//c`	`LC00:`
	`LDR X2,[X3]//e`

```
Allowed: 1:X0=1; 1:X2=0;
```

(a) Litmus test source



(b) “Flat” operational model state



(c) Axiomatic candidate

Fig. 1. An Arm litmus test (MP+dmb.sy+ctrl), with Thread 0 stores to x and y separated by a barrier, and Thread 1 loads of y and x separated by a control dependency from a conditional branch. 1b depicts a state of the “Flat” operational model, with per-thread trees of in-flight instruction instances. As indicated by the blue progress bars, each has executed more or less of its intra-instruction semantics. In this state, all are still uncommitted, and one instance of the second load has read from the initial memory, despite the branch being unresolved. 1c shows an axiomatic-model candidate execution. This execution is allowed, despite the cycle shown, as the read-read control edge is not included in the Arm axiomatic model observed-before cycle check.

2 Context and Overview

Modern architecture specifications are intimidatingly large and complex: the current Arm specification document is around 15 000 pages [13], which is challenging even for some pdf viewers, let alone semantics and mechanisation. What makes the current work feasible is the fact introduced above that the architecture semantics can be factored into ISA semantics and concurrency models. The two have different kinds of complexity. Full-scale ISA definitions are very large by the standards of formal artefacts: around 400 000 LoC for Arm. They can be more intricate than one might expect, to handle all the details of multiple privilege levels, address translation, exceptions, etc.; even simple register-to-register `add` instructions can involve pages of pseudocode. But this detail has been captured reasonably well, at least as far as sequential execution goes, in the Arm and RISC-V mechanised ASL and Sail ISA models. Concurrency models are typically more subtle, but much smaller, than the ISA semantics. Both Arm and RISC-V have developed models for substantial parts of the architecture, included in their documentation, but there remain open research questions about other aspects.

The basic idea of this factorisation is not new – it dates back at least to Sarkar et al. [60] and is to some extent implicit in earlier relaxed-memory research – but it can still be surprising. In a sequential or sequentially consistent semantics, one would typically think in terms of a machine state, including the register state for each hardware thread and a global memory state, and define the semantics of each instruction in terms of changes to that state, totally ordered in execution traces. That is unfortunately not a sound model of the relaxed behaviour of modern processors, which expose to the programmer some effects of their sophisticated internal microarchitectural optimisations, including out-of-order and speculative execution.

Capturing the architectural intent, to precisely specify the envelope of behaviour that a relaxed architecture allows, requires semantics with a quite different structure, that more loosely constrains the writes that reads can read from. Different styles of relaxed model and different tools do this in various ways. Operational models in an abstract microarchitectural style explicitly model out-of-order and speculative execution, while abstracting from most other hardware-implementation detail [59]. They are defined as transition systems over abstract-machine states which include, for each hardware thread, a tree of its in-flight and committed instruction instances, as shown in Fig. 1b.

There can be many in-flight instructions executing at once, out-of-order except where constrained by dependencies and barriers, and speculatively until previous branches have been definitively resolved – subtrees and instances are discarded or restarted if needed. Recent models for multicopy-atomic Arm and RISC-V follow the "Flat" model of Pulte et al. [50], in which this thread-state structure is linked to a flat underlying memory. Axiomatic models abstract further [4, 10, 29]. They are expressed as predicates over execution graphs of candidate complete executions, of the memory events and various ordering relations over them, as in Fig. 1c. A candidate execution is permitted if it is locally consistent with the instruction semantics and if some global axioms are satisfied. These are now typically expressed in the relational algebra style of Alglave et al. [10], e.g. requiring that some happens-before order is acyclic. Promising models [51] process instructions in program order but with explicit views of timestamps, and promises of future writes.

Early concurrency model exploration tools developed ad hoc handwritten semantics for the small fragments of the ISA used in simple litmus tests [5, 10, 59, 60]. As one scales up to cover more of the architecture, hand-writing a custom instruction semantics becomes impractical. Armstrong et al. [15], following Gray et al. [31] and Flur et al. [25], developed the isla-axiomatic model tooling that uses full-scale ISA semantics in Sail, both for Arm, automatically translated from the Arm-internal ASL, and for RISC-V, now developed by RISC-V International. Work is in progress in Arm to support a new version of ASL (with a more precise definition [12]) directly in herd [55].

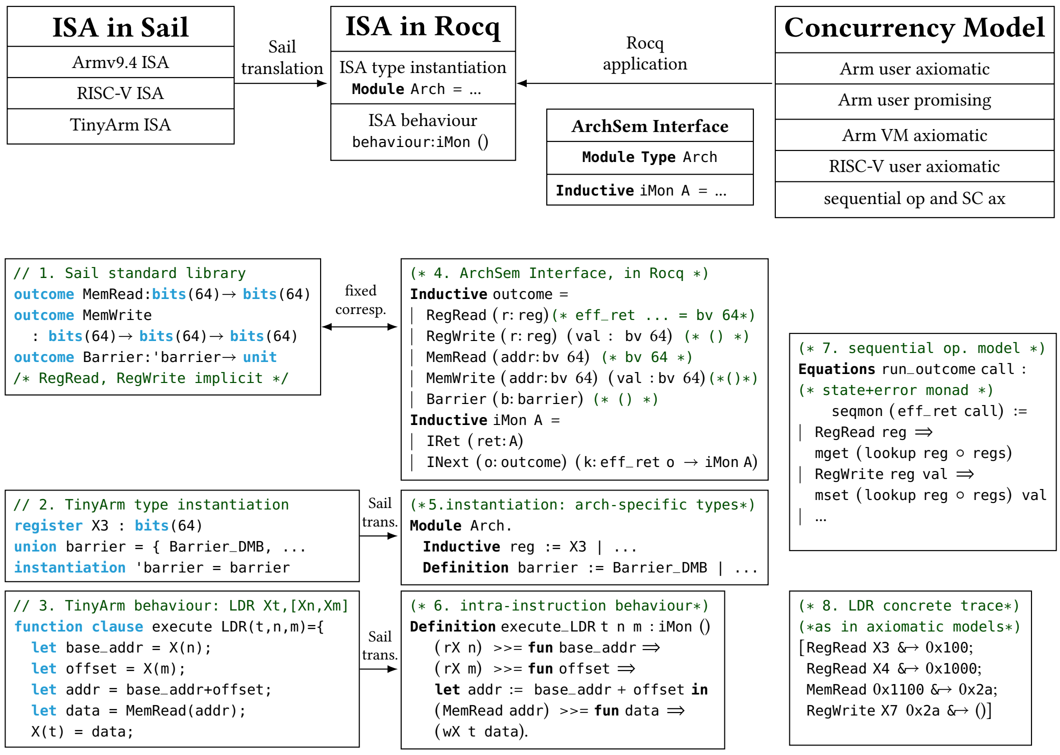
Sail and ASL are essentially first-order imperative languages, that let one define instruction semantics in terms of their register and memory effects (as in Fig. 2.3): reads and writes of general-purpose registers `X(n)`, and the `MemRead(addr)` read of memory.

The key point for this paper is that, as far as the instruction semantics in isolation goes, these are *uninterpreted* effects: their meaning is only provided by the integration with a concurrency model, which (one way or another) constrains the values that register and memory reads can see. The collection of these effects (and their intra-instruction ordering) thus form the essential interface between ISA semantics and concurrency model. This is a pleasingly narrow and relatively stable interface, that lets one work on each side largely independently: one can change the concurrency model without (usually) needing substantial adaption of the ISA semantics, and one can do modular proofs about concurrency models over an arbitrary ISA model, and vice versa. It can also be made largely, though not completely, architecture-generic. The ISAs of each architecture differ substantially, and some aspects of the interface do too (e.g. the kinds of memory barrier), but all share notions of register and memory effects, and the structure of axiomatic models can be reused.

We sketch the high-level structure of ArchSem at the top of Fig. 2. This is highly simplified to first explain the overall shape; later in the paper we explain the many ways in which tackling production architectures has required the actual structure to be elaborated.

One starts with an arbitrary ISA definition in Sail, written above a Sail version of the interface that gives the Sail types of the uninterpreted effects, such as `MemRead` (Fig. 2.1). The ISA definition also defines various types that the interface is parameterised on, e.g. the types of memory access kinds (plain, release, etc.) and memory barrier kinds (Fig. 2.2). For example, the Sail definition of the TinyArm `LDR` load instruction is at the bottom left (Fig. 2.3), eliding its decoding and register accessor functions.

From the complete Sail definition, an automatic translation from Sail to Rocq generates (1) Rocq types for the type instantiation of the interface (Fig. 2.5), and (2) a Rocq definition of the intra-instruction behaviour (Fig. 2.6). The former is packaged as a Rocq module that implements a module type `Arch` on which all ArchSem architecture generic code is parameterised. The latter is an instance of a free monad `iMon` over a type of algebraic effects, the `outcomes` sketched in Fig. 2.4. It abstracts from the Lem/OCaml `outcome` type of Gray et al. [31].



```
// 1. Sail standard library
outcome MemRead:bits(64)→ bits(64)
outcome MemWrite
  : bits(64)→ bits(64)→ bits(64)
outcome Barrier:'barrier→ unit
/* RegRead, RegWrite implicit */
```

```
(* 4. ArchSem Interface, in Rocq *)
Inductive outcome =
| RegRead (r: reg) (* eff_ret ... = bv 64*)
| RegWrite (r: reg) (val : bv 64) (* () *)
| MemRead (addr: bv 64) (* bv 64 *)
| MemWrite (addr: bv 64) (val : bv 64) (*()*)
| Barrier (b: barrier) (* () *)
Inductive iMon A =
| IRet (ret: A)
| INext (o: outcome) (k: eff_ret o → iMon A)
```

```
(* 7. sequential op. model *)
Equations run_outcome call :
(* state+error monad *)
    seqmon (eff_ret call) :=
| RegRead reg ⇒
  mget (lookup reg ∘ regs)
| RegWrite reg val ⇒
  mset (lookup reg ∘ regs) val
| ...
```

```
// 2. TinyArm type instantiation
register X3 : bits(64)
union barrier = { Barrier_DMB, ...
instantiation 'barrier = barrier
```

```
(*5.instantiation: arch-specific types*)
Module Arch.
  Inductive reg := X3 | ...
  Definition barrier := Barrier_DMB | ...
```

```
// 3. TinyArm behaviour: LDR Xt,[Xn,Xm]
function clause execute LDR(t,n,m)={
  let base_addr = X(n);
  let offset = X(m);
  let addr = base_addr+offset;
  let data = MemRead(addr);
  X(t) = data;
```

```
(* 6. intra-instruction behaviour*)
Definition execute_LDR t n m : iMon () 
   (rX n) >>= fun base_addr ⇒
   (rX m) >>= fun offset ⇒
   let addr := base_addr + offset in
   (MemRead addr) >>= fun data ⇒
   (wX t data).
```

```
(* 8. LDR concrete trace*)
(*as in axiomatic models*)
[RegRead X3 &→ 0x100;
 RegRead X4 &→ 0x1000;
 MemRead 0x1100 &→ 0x2a;
 RegWrite X7 0x2a &→ ()]
```

Fig. 2. Overview. This gives a high-level view of ArchSem and its instantiations, simplified for presentation; later in the paper we explain various ways in which tackling production architectures requires the actual ArchSem definitions to be more elaborate. The top shows the overall flow: an ISA definition in Sail is translated into a Rocq `Arch` module used to instantiate the ArchSem generic interface types, and a `behaviour` value of the `iMon` monad type defined by the interface. A concurrency model is applied to these to give an architecture model. Below we sketch some of this for the TinyArm ISA. 1 shows the Sail standard library declarations of primitives for reading and writing memory, and for memory barriers. Together with the implicit register access primitives, these are in a fixed 1:1 correspondence with the `outcome`s of the interface in 4 (which uses a function `eff_ret` that gives the return type of each outcome). 2 and 3 show some Sail TinyArm types, of register names and barrier kinds, and the behaviour of the `LDR` instruction, which are translated by Sail into 5 and 6 respectively in Rocq: an `Arch` module of type definitions and a value of type `iMon ()` defining instruction behaviour in terms of the generic `outcome`s (instantiated with the `Arch` types). The figure only shows an execute function, but the ISA models also define decode and, for the full ISAs, instruction fetch and address translation. 7 shows part of a simple sequential operational model, interpreting the `iMon` effects in a state+error monad, and 8 one concrete event trace of the TinyArm `LDR` instruction; such traces are used to construct the events of axiomatic-model candidate executions.

To define `iMon`, ArchSem implements a generic free monad library over arbitrary uninterpreted effects. Intuitively any effectful computation from `A` to `B` can be represented by a function taking an `A` and returning either a `B` or a pair of a call to an effect and a continuation – a (recursively) effectful computation from the effect return value to `B`. Depending on whether one wants to represent non-termination, this informal "recursively" could be inductive or co-inductive: either free monads or (with some extra subtleties) the itrees of Xia et al. [70]. In ArchSem we use free monads because we know the semantics of individual instructions in isolation should never diverge. Formally we represent effectful computations over effects `Eff` with types `A → fMon Eff B`, where:

```
Inductive fMon (Eff : Type) {B : Type} :=
| Ret (ret : B)
| Next (call : Eff) (k : eff_ret call → fMon).
```

We use an effect representation similar to that of McBride [43], but differing from the itree library [70], to avoid many Rocq universe issues that would otherwise crop up: `eff_ret` is a typeclass overloaded function associating a return type to any effect call value.

Concrete instructions only use our fixed set of effects `outcome` and do not take any argument data or return results, so can be thought of as effectful computations of type `fMon outcome ()`. Later we will extend this to handle non-determinism (§3.2), but for now one can think of the `iMon ()` type of instruction semantics as just `fMon outcome ()`.

On the other side of the interface, a concurrency model consumes a value of type `iMon ()` and defines the allowed behaviour of arbitrary initial machine states. Different styles of concurrency model consume the `iMon ()` object in different ways, which impacts the interface design. For example, simple operational models – such as sequential or promising models – generally just interpret the effects in some non-free monad, such as the code in Fig. 2.7. Abstract-microarchitectural operational models have states with many instruction instances in flight (each with an instruction-semantics continuation), with model transitions that progress or restart instances. The instruction semantics must thus be able to quickly compute the next possible outcomes from a partially executed state; and to have bounds on the remaining effects an instruction can do (cf. §7.2)

Axiomatic models work by defining validity predicates over *candidate executions* which are concurrent whole-program execution graphs (§5.2). The latter comprise the events of all the concrete traces of each instruction instance constituting the execution (e.g. Fig. 2.8), and relate these events with communication data, e.g. specifying which load read from which store, and how stores are ordered by coherence. Leaving mixed-size aside (described in §5.3), each outcome call, paired with its return value, gives rise to one `iEvent`, and intra-instruction traces are essentially just lists of `iEvent`, where `iEvent = fEvent outcome` and:

```
Record fEvent Eff = {call : Eff; fret : eff_ret call}. (* call &↦ fret *)
```

Modern architectures typically do not have a notion of termination (the days of the EDSAC instruction `Z` *stop and ring the bell* [69] are long gone), and ideally one would support reasoning about non-terminating machine executions. However, previous work on microarchitectural operational models typically covers this simply as infinite traces of their transition system, without any fairness considerations, while previous work on axiomatic models typically only seriously considers finite execution graphs. Moreover, one wants to be able to compare the behaviour that different models permit for terminating litmus tests. The top-level ArchSem type of architecture model therefore defines, for any initial architectural state, and some termination condition, the set of allowed final architectural states. For `n` hardware threads, these consist roughly of the register state per thread and a memory state (we refine this definition in §5.1).

```
Definition archModel := ∀(n : nat) (t : terminationCondition n),
  archState n → set (option {f : archState n | terminated n t f})
```

Ultimately one would want a finer notion of observation that captures memory-mapped I/O, x86 ports, DMA, and external interrupts, with their associated ordering properties. A proper treatment of that will require investigation of SoC semantics, well beyond our current scope.

```
Module Type Arch.
  Parameters
  (reg : Type)
  (pc_reg : reg)
  (reg_type : reg→ Type)
  (reg_acc : Type)
  (addr_size : N)
  (addr_space : Type)
  (mem_acc : Type)
  (CHERI : bool)
  (cap_size_log : N)
  (abort : Type)
  (barrier : Type)
  (cache_op : Type)
  (tlbi : Type)
  (exn : Type)
  (trans_start : Type)
  (trans_end : Type).
End Arch.
```

(a) Arch-provided types

```
Record mem_req := {access_kind : mem_acc; address : bv addr_size;
  address_space : addr_space; size : N; num_tag : N}.

Inductive outcome :=
| RegRead (reg : reg) (racc : reg_acc)
| RegWrite (reg : reg) (racc : reg_acc) (regval: reg_type reg)
| MemRead (mr : mem_req) | MemWriteAddrAnnounce (mr : mem_req)
| MemWrite (mr : mem_req) (value : bv (8 * mr.(size)))
   (tags : bv mr.(num_tag))
| Barrier (b:barrier) | CacheOp (cop:cache_op) | TlbOp (t:tlbi)
| TranslationStart(ts:trans_start) | TranslationEnd(te:trans_end).
| TakeException (e : exn) | ReturnException
| GenericFail (msg : string)

Instance eff_ret (call : outcome) := match call with
| RegRead reg _ ⇒ reg_type reg
| MemRead mr ⇒ result abort (bv (8*mr.(size)) * bv mr.(num_tag))
| MemWrite _ _ _ ⇒ result abort ()
| GenericFail _ ⇒ Empty_set | _ ⇒ () end.
```

(b) The outcome interface

Fig. 3. The full ArchSem interface (eliding typeclass parameters and classification functions)

3 The Full Interface

Modern CPU architectures involve many aspects that simplified models can (and did) gloss over. The full generic ArchSem interface is shown in Fig. 3, with the outcome effects in Fig 3b, parameterised by the architecture-provided type-level parameters in Fig 3a. We explain the details of this, and the reasons for the design choices, as we go. The interface is significantly more involved than the simplified version of the previous section, but it is still at a manageable scale, especially when compared with the ISA and concurrency models that provide and consume the interface.

Registers. Each architecture defines many registers, including general-purpose registers, floating-point and vector registers, and (in Arm) thousands of configuration and system registers. To a first approximation, these constitute the programmer-visible per-hardware-thread architectural state, though in the relaxed setting one cannot think of a per-thread single register-file state.

Each register has a type: many are bitvectors of particular lengths, but there are also booleans, structures, vectors, and so on. To encode this in Rocq we use an architecture-specific type `reg` of register identifiers – a big enumeration – and a function `reg_type : reg → Type` that defines the type contained in each register. Both are generated automatically from Sail's `register` declarations. The `RegWrite` outcome thus takes a value of type `reg_type reg`, and the `RegRead` outcome returns a value of that type, as defined by the `RegRead` clause of `eff_ret`.

The semantic granularity of registers is an architecture design choice which is exposed by relaxed execution. For example, the Arm `PSTATE` collects various process state information, such as condition flags and exception masking bits, but these are semantically independent, in the sense that a write to one followed by a read to the other does not entail a dependency (allowing hardware implementations to do register renaming for them separately) [31]. They thus should be separated into different registers at the interface. Architectures often also introduce aliases for parts of registers. For example, Arm assembly uses `Wn` and `Xn` for 32- and 64-bit references to the underlying 64-bit general-purpose register `Rn`. The latter are the semantic register names that should appear at the interface; the instruction semantics for 32-bit instructions does 64-bit accesses to them, bit-slicing or extending explicitly as appropriate.

Additionally, in Arm, system registers can be accessed *directly* through `MSR` and `MRS` instructions, or *indirectly* at other points in the instruction semantics that refer to them. Those might have much weaker semantics than expected: a register read might not read from the most recent corresponding register write. The distinction is represented by the architecture-provided register access kind type, `reg_acc`, which is added as a parameter to the `RegRead` and `RegWrite` outcomes. Normal accesses will use an underscore for this in the rest of the paper.

Memory. Memory read and write accesses take a memory request `mem_req` including an address, simply a bitvector of an architecture-provided `addr_size` – the maximum physical address size for full models, or a virtual address size for user-only models. Arm distinguishes Secure and Non-secure accesses, expressed by an `address_space`; for other architectures this will be unit.

Each request specifies a `size`, in bytes. The interface memory accesses are the architectural single-copy-atomic accesses. The data of a successful `MemRead` or `MemWrite` then has type `bv (8 * mr.(size))`, a light use of dependent typing that we find perfectly manageable in Rocq in practice. The interface also supports CHERI tags, with `num_tag` (cf. *CHERI* later in this section).

Memory reads and writes can fail due to a physical memory error, so the outcome return type also allows an `abort` error, describing the failure. On Arm, the ISA semantics propagates those as SError architectural exceptions. Exclusive access failures are handled differently, discussed below.

Each architecture defines many kinds of memory access, including concurrency modifiers, such as release, acquire, and read-modify-write (RMW), and identification of system accesses such as instruction fetches and translation-table-walk reads. We encode this in the architecture-provided `mem_acc` type. To allow architecture-generic reasoning where possible, the architecture must also provide some classifier functions of type `mem_acc → bool`, such as `is_rel_acq_rcsc` or `is_ifetch`.

Instruction fetch. Section 2 glossed over instruction fetch and decode and suggested that there is an `iMon ()` value per instruction. In reality, instruction fetching and decode is an important part of the ISA model, which contains the initial PC register read and ifetch memory read of each fetch-decode-execute instance, and specifies what happens in error cases. To make this more concrete, we show below a highly simplified outline of a single-instruction slice of the (400k line) Arm instruction semantics, as translated from ASL into Sail.

```
1  function __TopLevel() =
2    // in __FetchInstr:
3    if pc[1..0]!=0b00 then AArch64_PCAlignmentFault()              // check alignment
4    opcode = AArch64_MemSingle_read(pc, 4, CreateAccDescIFetch())   // read memory
5    // in __DecodeA64:
6    match opcode
7      0b1 @ (_ : bits(1)) @ 0b11100101 @ (_ : bits(22)) ⇒
8        // the semantics for one family of instructions, including loads LDR Xt,[Xn], from
9        // execute_aarch64_instrs_memory_single_general_immediate_signed_post_idx(n,t,...)
10       let address = X_read(n, 64)                                 // read register n
11       let data : bits('datasize) = Mem_read(address, DIV(datasize,8)) // read memory
12       X_set(t, regsize) = ZeroExtend(data, regsize)               // write register t
```

The PC register read and instruction fetch are done by the `AArch64_MemSingle_read` call on line 4. This auxiliary function does address translation (which might involve many additional accesses) and finally calls the Sail outcome for a single-copy-atomic physical memory read. The definition then decodes the opcode, pattern-matching on its binary shape, with a leaf clause for each family of instructions. Here we show a simplified form of a family of load instructions. It involves register reads and writes (`X_read` and `X_set`), primitive register accesses that get translated into `RegRead` and `RegWrite` in Rocq, and another auxiliary function `Mem_read` for a potentially non-single-copy-atomic memory read. This splits the read into multiple accesses if need be, does address translation for each, and again calls the underlying Sail outcome for physical memory reads.

The ArchSem translation of this into Rocq thus defines a single value of type `iMon ()` covering the entire ISA model. This defines the behaviour of arbitrary top-level fetch-decode-execute instances, not a collection of instructions with behaviour for each. Indeed, what counts as an instruction with certain parameters, versus a family of related instructions, is a matter of presentation, not a fundamental choice. The Arm ISA semantics does not define an abstract syntax type of instructions, while the RISC-V semantics does; ArchSem handles both uniformly.

On the concurrency model side, instruction fetches are not necessarily coherent with data memory accesses, without explicit data and instruction cache clean and invalidate instructions, such as Arm's `DC` and `IC` [64]. These give rise to `CacheOp` outcomes, taking an architecture-provided `cache_op`, which the concurrency model uses as required.

Virtual memory. Hardware virtual memory support translates the virtual addresses used in most code to the physical addresses that index physical memory. This translation is defined by page tables, in-memory tree-like data structures, and various control registers, all managed by an operating system and/or hypervisor. The ISA semantics defines the translation in terms of translation walks that do register and memory accesses as required, e.g. the Arm `AArch64.TranslateAddress` and RISC-V `translateAddr` auxiliary functions. For example, in a single-stage translation regime, `AArch64.TranslateAddress` takes a virtual address and various access type information, reads the physical address for the root of the tree from a `TTBR_ELx` register, and reads from the tree down to either a leaf entry, determining the physical address, or a failure, typically indicating that the virtual address is not mapped.

From a concurrency point of view, all this is special: information from translation walks, both partial and complete translations, can often be cached in Translation Lookaside Buffers (TLBs), and translations can often be done out-of-order and speculatively w.r.t. the instructions that use them. Externally this appears as additional forms of relaxed behaviour, captured axiomatically with loose constraints on the writes that translation reads can read from [6, 63], not with explicit TLBs. TLBs are not coherent caches, so software has to maintain them with explicit invalidation operations such as Arm's `TBLI` instruction. The `TlbOp` outcome makes those visible to concurrency models.

Our interface thus has to identify the translation-walk register and memory accesses and associate them to the other accesses that use them. We surround the part of the translation function that can be cached with `TranslationStart` and `TranslationEnd` outcomes. Everything between is considered TLB-cacheable, whether it is registers or memory. The architecture-specific `trans_start` contains the address and size to translate, as well as any context information required to look up an entry in the TLB, such as process identifiers (ASID for Arm). The `trans_end` contains either a failure or the physical address resulting from the translation as well as any associated attributes (device memory, cacheability, etc.). To associate a memory access with the translation it is using, one could introduce translation identifiers, but in our current ISA models, the translation always immediately precedes the access in the intra-instruction semantics. Multiple translations from misaligned accesses or load/store pair require them to be unordered with each other, as we discuss in §7.1.

Architectural exceptions. Like virtual memory, the instruction semantics captures much of the sequential behaviour of exceptions. For example, if the above fetch-decode-execute PC alignment check fails, the `AArch64_PCAlignmentFault` function will, among many other things, update the `PSTATE.EL` exception level register, read the appropriate vector base address register (`VBAR`), and update the PC to the appropriate offset from that. But exception entry and return also impact the allowed concurrency behaviour [13, 62], so have to be exposed in the interface. We add outcomes `TakeException` and `ReturnException` for this. When taking an exception, the ISA model must identify in an architecture-specific `exn` why the exception was taken; returning from an exception is just a marker that does not need a payload.

CHERI. CHERI architectures such as CHERI RISC-V [68], CHERIoT [11], and Arm's Morello [17], augment memory and memory accesses with tags that identify valid capabilities, one bit of tag every $2^{\texttt{cap\_size\_log}}$ sized and aligned bytes. Memory and tag accesses are single-copy atomic together, and if the tag footprint is smaller than the memory footprint on a write, e.g. for non-capability writes, all remaining tags are cleared.

3.1 Ordering and Dependencies

Relaxed CPU architectures like Arm and RISC-V guarantee that certain *syntactic dependencies* in the instruction stream order memory events, such as certain address, data, and control dependencies. Each arises from one memory-access instruction and can pass through multiple register-to-register and branch instructions, before reaching the final memory instruction. They thus involve both *intra-instruction* data-flow and control-flow, and the *inter-instruction* communication via registers. The former are determined by the instruction semantics, and are then used by concurrency models along with the latter to enforce the required ordering.

Arm and RISC-V prohibit (non-side-channel) observable value speculation, so intra-instruction data-flow must be respected. Most intra-instruction control-flow should also be respected, for example conditionals that decide whether to set flag values. There are a very small number of intra-instruction conditionals that have to be weaker to allow architecturally-permitted speculation, for example the condition that chooses which register to read from in an Arm `CSEL` conditional select, and the condition that decides whether a `CAS` compare-and-swap succeeds or fails. From the other direction, there are several places where the obvious intra-instruction sequential execution order of the existing ASL/Sail code (which includes all intra-instruction data and control relations) would be too strong: for instructions that do multiple memory accesses, register write-backs, and, for memory writes when translation is enabled, the ordering of the address translation and the data register reads.

For ArchSem, we need to understand:

(1) how the information of what is respected should best be expressed in the existing instruction semantics (or some reasonable modification thereof); and
(2) how should it be represented in the ISA/concurrency interface.

For (1), we see two main options:

(i) start from all the intra-instruction data and control flow implicit in the ASL/Sail, but annotate some conditionals as speculatable, and add some mechanism to force additional dependency, e.g. for the memory read and write of a `SWP` swap instruction; or
(ii) start from the obvious sequential order, and annotate to de-order where required using async/await constructs.

These have complementary pros and cons. (i) is attractively light in annotation, but that may make it easy to unintentionally de-order events. (ii) requires more annotation, but we think acceptably so, and it makes these architectural choices usefully explicit. For exploration tooling, either would be fine for axiomatic and promising models. For operational models, (i) would make it expensive to compute the set of next-allowable outcomes, and would create many irrelevant interleavings. For reasoning above the model, we expect (ii) to be preferable because it has less interleaving.

For (2), we see three main options:

(A) If the ISA were represented with a deep embedding, then one could symbolically compute (or re-compute) the dependencies whenever required (as done with a symbolic interpreter for an early version of Sail [31]), but such a deep embedding would make the interface much wider and less tractable.

(B) In addition to the free monad structure, one could require an explicit function (generated from a separate analysis of the ISA specification) that for each opcode computes its register footprint. This is roughly what isla-axiomatic does [16], using its symbolic evaluation for that analysis, but it is hard to scale robustly to instructions with complex dependencies, or to make it foundational.

(C) Extend the free monad to expose the async/await constructs of (ii) (one could also define a conversion to this representation from (i)).

Ultimately, the best choice seems to be (ii) and (C), because (C) aligns to (ii), does not require a deep embedding, and does scale to any desired instruction execution graph. We describe a novel async/await monad to support (C) in §7.1 along with paper maths properties to support it: we have proved that it can represent arbitrary instruction execution graph shapes, with strict and speculative edges where needed, for the few instructions that require it, and have an algorithm for recognising such executions.

However, we have not yet implemented this async/await in Rocq. Perhaps surprisingly, for most common instructions, no async behaviour is required: the required graph *is* a linear trace and can be adequately represented using our free monad `fMon`, given minor patches to the existing ISA definition, described below and implemented in §4.2. This means that, until the async/await monad is implemented, we can soundly represent the whole class of *linearly-ordered* instructions with the free monad. This is enough to do interesting reasoning about concurrency models, like the theorem we describe in §6. Later we will be able to smoothly extend our free monad setup to the async/await without changing the semantics of linearly-ordered instruction.

Linearly-ordered instructions include most common instructions one would use in litmus tests (such as in Fig. 1) and in regular sequential programming: arithmetic instructions, branches (including conditional branches), barriers, and single-access memory instructions.

That said, some interesting instructions *cannot* be represented with the free monad. For Arm, those are: `CAS` (Compare and swap), `CSEL` (Conditional select), `SWP` (Atomic swap), and any instruction performing multiple memory accesses in a single instruction, such as unaligned loads and stores, `STP` (Store pair), and most scalable vector and scalable matrix (SVE and SME) instructions. These are important for relaxed concurrency in general, but not especially relevant for our virtual memory abstraction result, as their address translation is largely orthogonal to their concurrency.

Multiple register reads. Many instructions, in particular most arithmetic instructions, can read multiple registers before doing anything else. In theory there is no intrinsic ordering between those register reads, which one might think poses a problem, given we can only represent a total order between all instruction outcomes. However, if the register reads all feed into the same set of effects inside the instruction, then one can order them arbitrarily, as nothing from another instruction can be ordered between them. This allows us to linearise instructions such as `ADD X0, X1, X2`.

Memory writes. For memory writes, the address usually comes from different register reads than the data, and those should be unordered. One approach, previously used in operational models [24, 26], is to use a separate outcome that announces the address of an upcoming write without its data, `MemWriteAnnounce (_ : mem_req)`. Then any register read that feeds into the address needs to be ordered before this new outcome. The `mem_req` parameter must match exactly with the following write. Furthermore, since in all concurrency models data dependencies are strictly weaker than address dependencies, we can put all the data register reads in between the `MemWriteAnnounce` and the `MemWrite`. This means that all registers used for the address are also considered data dependencies, but this does not change the semantics of any known model.

Branches. Lastly, we define control dependencies using PC register write outcomes: any operation before the PC write affects its value and therefore creates a control dependency to all later instructions as they all read the PC. However, if we treated the PC as a regular dependency this would create a strong data dependency. Therefore, we treat the PC specially (identified by the `pc_reg` parameter) by making PC register reads speculative. A PC read (that happens at the start of every instruction) can thus read a speculated value, without waiting for the previous instruction's PC write (a `MemWrite` will still have to wait for that previous PC write though, in order to respect the control dependency).

This means that control-flow branching instructions can be linearly ordered by first reading the PC, then reading whatever is needed to make a branching decision, and finally writing the new PC. Non-branch instructions must write the PC as soon as possible after decoding, to not create unwanted dependencies into the PC. This is not how current ASL or Sail specifications are written, so modifications are required to be compatible with ArchSem (see Section 4.2).

3.2 Intrinsic Non-Determinism

ISA semantics do explicit non-deterministic choices in many places. The most obvious example is store exclusives, that can fail non-deterministically, or succeed in certain conditions. There are also places where register or memory values are given `UNKNOWN` values, e.g. when taking exceptions.

We therefore need our instruction monad to support non-determinism, making it closer to the choice trees from Chappe et al. [20], while still being terminating. With terminating structures, this can simply be obtained by adding a new effect to the free monad instead of a separate type:

```
Inductive MChoice := ChooseFin (n : nat).      Definition eff_ret '(ChooseFin n) := fin n.
Definition cMon Eff := fMon (Eff + MChoice).   Definition iMon := cMon outcome.
```

Here `fin n` is type of integers between 0 and `n` − 1. `eff_ret` is also overloaded on type sums with the obvious semantics. In the definition of `cMon`, `MChoice` appears free, but it is actually always interpreted as a non-deterministic choice, representing the same behaviour as `BrD` in the ctree theory. We do not need a concept of invisible steps, because of systematic termination, so there is no need for another type of branching like `BrS`. On the other hand, we restrict ourselves to finite choice, ensuring that the problem of knowing whether a choice monad recognises a given trace stays decidable. The distinction between finite and infinite choice is only relevant with terminating choice monads, as one can create an infinite choice from a finite choice primitive using non-termination.

The fact that MChoice is actually interpreted appears in particular in the trace semantics of `cMon`. The generated sets of traces do not contain `MChoice` effects, instead, when checking a trace, any return value is possible. As a result, doing a 4-way choice or two 2-way choices in sequence give semantically equivalent `cMon` objects. Concurrency models should respect this equivalence relation and, for example, must not be able to distinguish between those two cases.

`ChooseFin 0` is an interesting operation, which we'll call a "discard" or NB (No Behaviour). It allows an ISA model to discard an execution path as being retroactively impossible. For example, doing a 2-way choice then a discard in one of the branches is equivalent to the fully deterministic behaviour of always taking the other branch. Similarly, when doing a `MemWrite` with an exclusive access type and the operation cannot be completed without breaking the exclusive rules, the whole execution is discarded by the concurrency model as if the ISA model issued `ChooseFin 0` instead.

This is correct because the ISA model for a store exclusive will previously do a non-deterministic choice of success or failure before calling the `MemWrite` outcome. More generally, ISA models should always have at least one behaviour (though that might be the `GenericFail` below that induces undefined behaviour), but we do not check or prove this property, other than by code inspection of the Sail/ASL specifications.

3.3 Failure and Undefined Behaviour (UB)

Usually one thinks of UB only for high-level and intermediate languages such as C, C++, and LLVM, where it allows optimisations to assume that the source program does not exhibit certain behaviour, by allowing arbitrary implementation behaviour where it does not. At the architecture level, a complete architecture specification should define bounded allowed behaviour for arbitrary initial states: even where there is some unpredictable behaviour, e.g. for Arm page-table break-before-make violations, it must be *constrained* unpredictable behaviour, otherwise there can be no security guarantees for execution at higher exception levels. However, one often wants to work with models that have some restricted scope, and which thereby can be simpler, or one has to, because some feature has not yet been formally modelled.

Historically, such scope restrictions have typically been implicit. For example, early work only considered simple non-mixed-size accesses and some limited forms of synchronisation, and later models gradually addressed more. In each case, the scope has been described just in prose around the model. Instead, our framework supports models that explicitly define the limits of their scope, by flagging undefined behaviour where executions go beyond it. For example, a non-mixed-size model should flag UB if there are any overlapping accesses that do not have identical footprint, and a non-instruction-fetch model should flag UB if there are races between instruction fetches and writes. Undefined behaviour can also arise from the ISA model, most simply for instructions that a particular ISA model does not cover, and more interestingly when using an ISA model simplified by symbolic execution (as in Islaris [57]), if the constraints assumed in that simplification are violated. A proper notion of UB enables statements and proofs of refinement between models of differing scope, e.g. for future proofs that a mixed-size model extends a non-mixed-size model.

The core ArchSem definitions are set up to handle UB, arising either from the ISA, where UB is signalled in the interface by the `GenericFail` outcome, or from the concurrency model. In abstract-microarchitectural operational models, identifying UB is straightforward: these build executions incrementally, and a partial execution is UB if it reaches either an ISA or concurrency-model UB arising from non-speculative instruction instances. Existing architecture axiomatic models, on the other hand, only define consistency of complete candidate executions, not built incrementally. Because Arm and RISC-V relaxed models allow cycles in program-order and reads-from, it is not obvious how to define the allowed executions incrementally. It is thus hard to handle UBs within individual instructions that prevent completion of the instruction and of the thread, such as a `GenericFail` for an unsupported instruction, or a read of an unmapped address in a user-only model, where there is no read value to complete the instruction.

This is a new problem that we identify in this work. We discuss in §7.2 how UB-aware axiomatic models could be defined.

4 Instantiating with ISA Semantics from Sail

Now that we have the interface in hand, we turn to instantiating it with instruction semantics from the authoritative full-scale ISA models. There are two aspects here: the *interface types* (the types that the interface is parametric on, `reg`, `reg_acc`, `mem_acc`, etc.), and the instruction semantics behaviour. The interface types are part of the interface signature, and therefore, any changes might require adapting the concurrency model code. But the internals of the behaviour definition are abstract as far as the concurrency model definitions are concerned, and therefore can be changed without impacting the concurrency models. This is a significant practical simplification: the interface types for Arm are only around 400 LoC, plus around 1500 register definitions, compared to the 400 000 LoC of the complete Sail model.

ArchSem has been successfully linked with three ISA models so far:

- `sail-arm`: an Armv9.4-A model automatically translated from the official Arm ASL, updating an earlier Armv8.6-A model and automated ASL-to-Sail and Sail-to-Rocq translators [15], and then manually adapted to fit the interface. The manual adaption required is relatively minor: three files changed, with 115 insertions and 86 deletions.
- `sail-riscv`: the official RISC-V formal model, again adapted (+133/-34 lines).
- `sail-tiny-arm`: a toy 5 instruction model with handwritten instruction semantics, but the interface types imported from `sail-arm`, so that they expose the same interface signature.

All these models can run inside Rocq in ArchSem with the sequential model, with `vm_compute`. They run at about 1 instruction per second for `sail-arm`, and 80 for `sail-riscv`, for factorial functions combined from C, with system register values that turn address translation off. This is sufficient to be of practical use in debugging and evaluating our framework.

4.1 The Sail to ArchSem pipeline

ArchSem expects the interface types to be provided as a Rocq module of an `Arch` module type. The ArchSem architecture-generic code is written in functors that depend on that.

We added an explicit **`instantiation`** declaration to Sail so that each ISA model can specify its own interface types, sizes and the classifier functions for architecture-specific memory access types. Correspondingly, the Sail standard library was extended with **`outcome`** declarations, which define the possible events that an ISA model is allowed to instantiate. We allow these outcome declarations to be parameterised by arbitrary types, including type-level integers and constraints, as permitted by Sail's dependent type system. This ensures we are flexible enough to precisely represent any architecture specific details, and enables us to use correspondingly precise types in the Rocq translation. This system for defining architectural effects is now used across all Sail backends, including direct compilation to an executable emulator (via C or OCaml). The **`outcome`** statements can only appear in the Sail standard library, but having them as first-class language constructs means the set of events the concurrency model can interact with could, if necessary, be relatively easily extended, with some minor changes in Sail standard library and the Rocq and Isabelle backends, as well as ArchSem's `outcome` type.

The Sail-to-Rocq backend uses these declarations to create an instance of the `Arch` module, including generated typeclass instances for the types. We replaced the previous Sail-to-Rocq register access code, which was untyped and used strings for register names, with an enumeration of registers, and use Rocq's dependent types to check type correctness of register accesses.

We also made several improvements to the Sail-to-Rocq backend to improve performance: removing some experimental embedded proofs mirroring Sail's type system, using a custom decidable equality procedure generator to overcome known inefficiencies in Rocq's built-in generator, and improving the efficiency of the rewrite that ensures all pattern matches are exhaustive. These improvements were important for dealing with the increasing size of ISA models as new architectural features are added.

4.2 Sail-Arm Modifications

We made a branch of the Sail-Arm model that implements the dependency scheme described in Section 3.1. To manage the large scale of the Arm instruction set, the ASL model source makes extensive use of common helper functions, especially for memory accesses, which provide localised places where our adaptions can be applied. To get the desired ordering for memory writes, we added copies of the memory write functions, which take the register name as an argument rather than the data, and perform the read after announcing the address, truncating the data to the size of the write. The ordinary copies of the write functions now trigger an assertion to indicate that the instruction is unsupported, which is preferable to generating false dependency information.

Similarly, we altered the points at which PC register writes occur according to Section 3.1. A new helper function performs a normal PC update to the next instruction, and is placed at the start of every instruction that does not branch or fault; in memory access helper functions once we know that the access will succeed; and in the branching helper functions in the case the branch is not taken. The model may read the PC multiple times, possibly after we have updated it, so we also renamed the register, keeping the original name as an intra-instruction global variable for the PC value at the start of the instruction. We add a flag to indicate whether the PC has been updated, to detect unsupported instructions which have not been updated to the new scheme.

We patch the `AArch{32,64}_TranslateAddress` and `AArch{32,64}_AT` functions in the Sail-Arm model to add `TranslationStart` and `TranslationEnd` markers. For the payload of the translation end markers, we instantiate the result type of the interface with the existing `AddressDescriptor` type of the model, which contains the output address (along with permissions and attributes) or a fault. For the translation start maker, we define a new record type `TranslationStartInfo` containing the VA and size of the access, as well as relevant information about the translation configuration, e.g. the VMID and ASID. We obtain the latter using existing functions like `AArch64_GetS1TLBContext` (originally used in the specification for modelling a TLB), which determine the translation configuration by reading the correct system registers.

4.3 Sail to Isabelle

We have ported a version of the concurrency interface to Isabelle, and adapted the Sail-to-Isabelle translation to generate definitions targeting it. We use these definitions to prove key properties of the Arm architecture required for the VM abstraction theorem we present in §6. Doing this in Isabelle allows us to adapt and reuse Isabelle libraries and tooling developed by previous work [17] that has shown the feasibility of proving a whole-ISA security property of the Morello CHERI-Arm architecture in Isabelle. Attempting to do this proof in Rocq would have required a prohibitive amount of effort for us to redevelop the tooling and automation required to handle a full ISA specification with hundreds of thousands of lines in Rocq ([17] reports around 2 person-years of effort on developing tooling and automation, and another 2 person-years for the whole-ISA proof itself). We describe in §6 how we combine the Isabelle proofs of key ISA properties with Rocq and pen-and-paper proofs about the combination of Arm9.4-A instruction semantics and memory models. This is enabled by the fact that the Isabelle and Rocq proofs use essentially the same interface, with the minor differences mostly due to limitations of Isabelle's type system. In particular, we cannot have an effect-generic free monad, so we monomorphise `iMon` and split the `Next` constructor into one per effect, e.g. `Read_reg : reg → (reg_value → iMon A) → iMon A`. Apart from such technical differences, the definitions are similar enough to check equivalence by manual inspection.

5 Linking with Concurrency Models

Now that we have defined our ISA interface and instantiated it from Sail models, we can focus on building concurrency models above it. A concurrency model is an object that, when combined with an ISA model, produces a full architecture model.

5.1 Architecture Models

In Section 2 we showed a simplified definition of architecture models. The full definition is:

```
Inductive archModelResult (flag : Type) (n : nat) (termCond : terminationCondition n) :=
 | FinalState (fs : archState n) (t : terminated n termCond fs)
 | Flagged (f : flag)
 | Error (msg : string)
Definition archModel (Set : Type → Type) (flag : Type) :=
   ∀(n : nat) (termCond : terminationCondition n),
      archState n → Set (archModelResult flag n termCond)
```

Informally, this means that an architectural model, for a given set type `Set` and a given type for special flag results `flag`, is a function that, for any number of hardware threads `n`, and termination condition `t`, takes an architectural state of `n` threads and produces a `Set` of either final states, unspecified results, or errors. This is parameterised on the `Set` type to allow executable (`list A`) or non-executable (`A → Prop`) sets (cf. *Model executability* below). Errors represent Undefined Behaviour (UB), which means that execution went outside the scope of the model. The string is not semantically relevant, but is there to explain the error. The `flag` type is meant to represent large but not unbounded sets of final states that can arise in some architectures. For example, in some situations, Arm has "Constrained Unpredictable" behaviour. In some cases one would need to precisely delimit the large set of allowed resulting states, but it may not be feasibly computable; representing this as a symbolic special *flagged* result allows us to compare executable models in the presence of such styles of architecture specification.

Architectural states of `n` threads are records of `n` partial register maps and a partial memory map:

```
Definition registerMap := dmap reg reg_type. (* dependent map based on gmap *)
Definition memoryMap := gmap (bv addr_size) (bv 8).
Record archState n := {regs : vec registerMap n; mem: memoryMap}.
```

The maps are partial to let one appropriately restrict the scope of the model execution. If an execution attempts to read or write any register or address that is not present, the model raises UB. This allows modelling real systems, where not all virtual or physical addresses are bound to physical memory, and some level of compositional reasoning.

A *termination condition* over `n` thread is just a predicate on the `registerMap` of each thread, typically just checking program counter values.

```
Definition terminationCondition (n : nat) := (* thread id *) fin n → registerMap → bool.
Definition terminated (n : nat) : terminationCondition n → archState n → bool := ...
```

To relate that to real hardware, we expect that at the termination point there is sufficient synchronisation code to observe an architectural state, as in the `litmus` test harness [7, 9, 60].

Note that there is no identified "program": for an architecture semantics that covers instruction-fetch and self-modifying code, the semantics just executes from an initial state, which includes in the register maps the initial PC value for each hardware thread. If users do not want to deal with instruction fetches, they can hard-code an instruction memory into the ISA model.

Model executability. ArchSem is designed to support both executable and non-executable models. For proof – our main focus in this paper – it is typically not important whether a model is executable, but for model exploration tools, it certainly is. One has to distinguish several related notions of executability: executability in principle versus in practice, checking versus enumeration of the model-allowed machine executions, and concrete versus symbolic execution.

ArchSem ISA models are always executable (in principle and in practice), in the sense that the next outcome of an instruction instance can be computed. However, ISA models with intrinsic nondeterminism, such as `UNKNOWN` 64-bit register values, can make it impractical to concretely compute all intra-instruction traces of an instruction instance even given concrete register and memory read values, as one would have to branch on those values. In some cases, e.g. when checking normal litmus tests, a reasonable workaround for this is to arbitrarily determinise most such choices (e.g. to `0`): in the Arm ASL many `UNKNOWN`s just represent as-yet-uninitialised values, which should never be used before initialisation; only some represent real architectural nondeterminism.

ArchSem supports both executable and non-executable concurrency models, which then lead to respectively executable and non-executable architecture models. Of the models in the rest of this section, only the sequential model is executable in a strong sense, that it can enumerate the model-allowed final states; it can run in practice on small examples inside Rocq, using `vm_compute`, with the above determinisation. It should also be possible to extract OCaml for external execution.

Our axiomatic models are executable in a weaker sense: they can decide the consistency of a candidate execution, but not enumerate the consistent candidates. This critically relies on our executable relation library that contains formally verified implementations of the usual relational operations (sequence, transitive closure, ...). Lifting this restriction would require a candidate execution generator inside Rocq – a verified but perhaps performance-naive analogue of tools such as memevents and herd. That is straightforward in principle but needs nontrivial engineering, e.g. to connect to existing litmus-test front-ends. Similarly, our promising model can decide whether a given memory trace is acceptable, but not currently generate them. This restriction could be lifted similarly.

Relating models. We also define *weaker* and *wider* relations between architecture models. A *weaker* model allows more behaviours on the same initial state, e.g. we expect the Arm user-mode relaxed memory model to allow strictly more behaviours than our Arm SC model. On the other hand, *wider* is about scope: a model is wider than another if it defines the behaviour of some initial states that were UB on the narrower model, while having the same set of behaviours on non-UB initial states. For example our Arm VMSA model should be wider than the user one because it will define behaviours on executions modifying their own page tables, which the user-mode model would flag as UB.

5.2 Axiomatic Models

As usual, axiomatic models are defined over *candidate executions*, which combine a *pre-execution*, containing the program-ordered events of each thread, and data defining the communication and synchronisation, including the reads-from relation specifying which writes each read reads from.

ArchSem defines pre-executions as pairs of an initial architectural state and, for each thread, a list of individual instruction traces (traces of the top-level fetch-decode-execute `iMon` value):

```
Inductive fTraceEnd Eff A := FTERet (a : A) | FTEOpenCall (e : Eff) | FTEStopped.
Definition fTrace Eff A := list (fEvent Eff) * fTraceEnd Eff A
Definition iTrace := fTrace outcome.
Record preExec n := {init : archState n; events : vec (list (iTrace ())) n}.
```

A pre-execution is *valid* for an ISA model if each instruction trace (`iTrace`) can be individually validated by the model. Each of those individual instruction traces can be *complete*, with a return value (`FTERet`), or *partial*, either ending with an open call or after a response was provided to the last call. We lift that distinction to pre-executions by calling them *intra-instruction complete* if each individual instruction trace is complete, or *intra-instruction partial* otherwise (if at least one instruction trace is partial). Events in the pre-execution are indexed by *event IDs* (`eid`), just giving their position in the above structure:

```
Record eid := {tid : nat;(* Thread ID *)
               iid : nat; (* Instruction ID *)
               ieid : nat; (* Instruction Event ID *)
               byte : option N (* Byte index, cf §5.3, assume None for now *)}.
```

As usual, a *candidate execution* adds a reads-from relation (`rf`), and a coherence order over memory writes to the same location (`co`). We also include the register reads-from relation `rrf`, and the `lxsx` relation relating matching load and store exclusives. These relations are expressed as `grel eid = gset (eid * eid)`, which is an executable relation data-structure derived from stdpp's `gset`, for which we have developed a library of proven-computable relational algebra operators.

```
Record candExec (mixed_size : bool) n := (* ignore the mixed_size flag for now, cf §5.3 *)
  {pre_exec : preExec n; rf : grel eid; rrf : grel eid; co : grel eid; lxsx : grel eid}.
```

We define all usual derived relations such as `po` and `fr` above these definitions.

Candidate executions are *well-formed* when all the relations have the obvious properties, for example `rf` must go from a memory write to a memory read with the same address, size, and value (ignoring mixed-size until §5.3). If a candidate is well-formed and intra-instruction-complete, a final `archState` can be extracted by taking the latest register write in the program order for each thread, and the latest memory write in `co` for each memory byte.

Differences. Our presentation differs from most previous definitions in two ways. First, we do not have events to represent initial memory. Instead, it is allowed for a memory read to not be in the range of `rf`, in which case it must read from initial memory, which is checked by the well-formedness property (and similarly for `rrf`). This reduces the number of events, which will be valuable in future for practical executability, and avoids the question of single-copy atomicity of initial writes, especially for non-mixed-size models. Second, we allow our `co` relation to contain more events than just memory writes. All events that can be directly observed by other threads – for example broadcast cache invalidations – should be in `co`, and we leave it up to the concurrency model to define which those are. For example, the VM axiomatic model orders `TlbOp` events in `co`.

ArchSem axiomatic models classify candidate executions into three kinds:

- Consistent and ok: this candidate is consistent. If it satisfies the termination condition, it gives a final state.
- Consistent and UB: this candidate is consistent and flagged undefined behaviour.
- Inconsistent: this candidate is inconsistent and can be disregarded.

The combination of this with an ISA model can be lifted to a non-executable architecture model by considering the set of candidate executions containing the initial state, that are well-formed, valid according to the ISA model, and consistent according to the axiomatic model. Those candidates produce a final state in each "consistent and ok" case (if the candidate is complete) and UB otherwise.

Concrete axiomatic models. We have written versions of five existing axiomatic models, for Arm and RISC-V, above ArchSem:

- A sequential model.
- A regular Arm user-mode model (non-mixed-size) based on the version of Pulte et al. [50].
- A mixed-size Arm user-mode model based on Alglave et al. [4], as a diff to the above.
- A virtual-memory (VM) Arm model based on the ESOP'22 model by Simner et al. [63].

- A user-mode RISC-V axiomatic model, from the RISC-V specification [67]

Much of this is reasonably straightforward, as it should be – mostly transcribing cat files into similar notation in Rocq (e.g. `A | B` becomes `A ∪ B`). However, there are still some significant gaps between a cat file and a fully defined axiomatic model. First the usual axiomatic model definitions typically do not define register-level coherence, since they assume a candidate execution generator, thus we have to add it. Since none of the above models support system register writes, this just means requiring register reads to read from the po-latest write. We therefore also have to classify any candidates with system register writes as UB. In addition, cat file models use various primitive sets of events such as `A`, `L`, `DMB`, etc. We need to use the interface types to define those consistently – simple for user-mode sets but less obvious for the classification of TLB invalidation instructions.

We also need to add more UB conditions to be sound: ISA models will issue instruction fetches and translation table walks, which need to be handled, and in the user mode models we require both to read from initial memory to avoid UB. In the VM model we keep that requirement on instruction fetches but relax it on translation table walks. For that model we also need to ensure that execution does not need mixed-size semantics, by raising UB if there are two memory accesses that have overlapping but not identical footprints.

Furthermore, in the presence of undefined behaviour at the level of ISA models (traces ending in `FTEOpenCall (GenericFail "...")`), it is possible for UB to exist only in an intra-instruction-partial candidate execution while not being observable in any intra-instruction-complete execution, a problem that was not considered before this work. At present to the best of our knowledge none of the Arm or RISC-V axiomatic models from the literature support defining the consistency of an intra-instruction-partial execution. Since our current axiomatic models are based on models from the literature they have the same limitation. We sketch how to lift this in §7.2.

5.3 Mixed-size Candidate Executions

Representing mixed-size candidate executions presents an additional complexity: for an aligned (single-copy-atomic) access, the ISA model will issue a multi-byte memory read outcome, but the official Arm mixed-size axiomatic model expects one memory read event per byte. This breaks the assumption we made so far that ISA events (`iEvent`) always exactly match axiomatic model events (`eid`). However the ISA model issuing a `MemRead` in a single block is semantically significant, because it declares the read is a single-copy-atomic (SCA) access, which would have different semantics if split per byte. We therefore give up the `1:1` mapping between `iEvent` and `eid`.

In order to split `MemRead` events per byte, we use the `byte` field of `eid` to allow multiple `eid` pointing to the same `MemRead` event. Each of those `eid` represent a different axiomatic model event. For all other event kinds, only the `eid` with `byte = None` is valid. Unlike Alglave et al. [4], we do not split writes, because we observe that, in their Arm model (and, we expect, other reasonable models), individual write bytes cannot be separated by the `ob` order, and minimising the number of events will improve tool performance in future.

This changes the wellformedness condition and eid indexing scheme of candidate executions in several minor ways, so we use the `mixed_size` type-level parameter to distinguish whether a given candidate execution should be interpreted as mixed-size or not. Mixed-size executions are used to define our mixed-size model, that otherwise follows [4].

5.4 Sequential Model

We wrote an operational Arm uniprocessor sequential model as a simple example concurrency model satisfying the previous shape – it just throws an error if there is more than one thread. This model can be used as a sanity check: any sensible Arm concurrency model should be *wider* than the sequential model, otherwise it is breaking sequential programs. It can also be used to validate ISA models experimentally on sequential tests.

The basic definition is straightforward: a state transition system where the state is just an architectural state (`archState`), and each transition runs an entire instruction, outcome by outcome. For all the register and memory outcomes it just does the immediate corresponding lookup or modification of the state; the other outcomes are no-ops.

However, even here there are some subtleties in how to implement such a model. If we only did the above, it would be unsound with respect to the architecture, due to various system aspects that create concurrency even within a single hardware thread, but following the ArchSem design, the model should identify the limits of its valid scope. First, the sequential model must forbid writes to system registers, because they have relaxed single-threaded behaviours. Second, since instruction fetches and translation table walks are both from non-coherent caches, we need to forbid them to read from modified memory. The model can achieve this by tainting any memory byte that has been modified since the initial state, and, if any of those two kinds of accesses are made from tainted memory, declaring UB. With those two concise modifications, the sequential model should be sound with respect to real system-level concurrency models. We can run this model against all our Sail ISA models on simple programs to check that our Sail import pipeline is correct.

As an initial exercise of the ArchSem definitions for proof, we proved in Rocq the operational sequential model's soundness w.r.t. the SC axiomatic model, for non-mixed-size executions. This uses a slightly older version of the framework. We made heavy use of our extensible automation tactics, which have been augmented alongside the proof; these substantially eased the proof.

5.5 Promising Model

We have also defined an ArchSem version of the user-mode promising model by Pulte et al. [51]. Our version is a non-executable model, though it is able to decide whether a given memory history is admissible by executing each thread independently against it.

Promising models are state transition systems with two kinds of transition: either promising a new write and adding it to the memory trace, or running an entire instruction on one thread. The various ordering constraints that appear in axiomatic models are represented with timestamps that refer to some global time. We implemented both a "non-certified promising" model, where promising transitions can be made in an unrestricted manner, but a promising trace does not produce a final state unless all promises have been fulfilled, and a "certified" version that defines the set of allowed promise from sequential evaluation of the model. For the latter we support a sound but restrictive concept of UB.

These instruction transitions are executable, which means that, given a memory trace, our model can compute its validity. This is achieved by interpreting the ISA monad against an effect handler, like the sequential model. However, this interpreter is more subtle: it tracks an instruction wide timestamp to represent the order imposed by `iio`, then read operations (register or memory) update the value (by doing a max), and register writes store this timestamp back into the state. For memory writes the handler tries to fulfil an existing promise, and adds a new memory write at the end of the trace otherwise. Barriers work as in the original model.

6 Virtual Memory Abstraction Theorem

To exercise ArchSem, we prove an important property about the Arm architecture that was not previously precisely stateable, let alone provable – specifically, about the combined Armv9.4-A ISA and the axiomatic relaxed virtual memory model of Simner et al. [63].

Architectures are designed so that system software, such as language runtimes, operating systems, and hypervisors, can use them to provide simpler and more constrained abstractions to user code.

For operating systems, one key abstraction that the OS and architecture together must provide is that of *virtual memory*. Arbitrary user code running in a user-mode process should behave as if it has a specific partial memory space, with some addresses mapped to physical memory but (except where specifically permitted) without access to the physical memory of other processes, or the OS itself. Moreover, it should behave as if running above a user-level relaxed concurrency model, without any of the relaxed-memory complexity of address translation accesses, TLBI invalidation, etc., and without any of the instruction semantics complexity of translation walks, beyond the extensional address translation function they define. The actual physical addresses used should not be observable. In particular, once the OS jumps to user-mode (EL0 in Arm), then, under some conditions on the virtual-memory control registers and page tables being set up properly, the EL0 program should behave according to the user-mode abstraction.

Note that this is a property solely about the architecture, that captures whether the virtual memory architecture is well-designed. It can be stated and proved without considering any specific OS – though the preconditions of the theorem should be established by an OS (and that fact verified by a verified OS). In future, one might hope for theorems like this to be reused, at the heart of verified stacks and providing important feedback to architecture design.

The important direction is soundness: that the system architecture, with constant well-set-up page tables, does not permit any behaviours of the EL0 execution that the user-mode model doesn't allow. This is critical for application-level programmers to be able to safely use the user-mode abstraction. The other direction, "completeness", would assert that all behaviours allowed by the user-mode model can be observed on a constant page table setup running on the system-level architecture. This would ensure that the user-mode model is not unnecessarily weak, but it is not important for safety, or for reasoning about user code. In this work we only prove soundness, although we are confident that completeness would also be achievable.

There could be many different versions of such a theorem, e.g. allowing controlled sharing between processes, or for processes to request and receive additional mappings for the OS. In this paper we prove a relatively simple form: for a specific class of system register and page table configurations that fix a constant mapping. In the absence of an Arm model that combines virtual memory, self-modifying code, and exceptions, we handle just the former and assume the absence of the latter two. Moreover, because the virtual memory model [63] does not support mixed-size, we assume the absence of mixed-size accesses. Under these assumptions, we show that any consistent system-level intra-instruction-complete execution can be lifted to a consistent and intra-instruction-complete user execution with a matching final state. We do not consider the case of intra-instruction-partial executions as their consistency is not yet defined, thus, in this section, any executions mentioned are always implicitly intra-instruction-complete.

As we have described, the definition of whole-architecture semantics involves challenges of different kinds, arising from the scale of the ISA semantics and the subtlety of the relaxed concurrency semantics. Precisely stating and proving this theorem has corresponding challenges, which we handle in correspondingly different ways.

The main challenge is establishing the necessary facts about the ISA: that (in an appropriate configuration) any memory access uses a physical address that arises from an address translation, that any such translation matches a functional characterisation of the ISA translation-walk definition, and various related properties. To prove these, we extend infrastructure from earlier work proving whole-ISA security properties of Sail models as exported to Isabelle [17]. That used an OCaml tool that does some analysis of a Sail ISA definition, and generates a topologically sorted list of Isabelle lemma statements and proofs, that all functions within the definition have the properties we need. Most are proven with a single tactic that does decomposition, along with some special logic to establish, for example, the well-bracketedness of translation markers. Only some of those proofs needed manual tweaking. This strategy and tooling makes the proof feasible despite the scale. The 430k LoC Sail ISA semantics is automatically translated to a 1.2M LoC Isabelle definition. The above tool generates 50k LoC of lemma statements and proofs, to which we had to add around 10k LoC of manual definitions and proofs. Running the proof takes around 3 hours (see §6.4).

The proof about the concurrency model is subtle, and would be very error-prone to do by hand, but is much smaller. The challenge here was more that of setting up the ArchSem and concurrency model definitions right, and in a form usable for proof, along with adequate proof automation, which made the proof robust in the face of changes to the definitions, e.g. to the axiomatic model.

This combination forces us, pragmatically, into combining results from multiple provers. In an ideal world, mechanised proof would be entirely within one system, and in principle one could explore whether that Isabelle proof automation is reproducible in Rocq, or develop an Isabelle version of ArchSem as a whole. The first might require multiple person-years, based on prior Isabelle experience, while the second would sacrifice the possibility of using the resulting definitions within Iris, and one would have to work around the lack of dependent types. Thus, regrettably, neither seem viable at this point. Moreover, the Isabelle and Rocq versions of the instruction semantics share the same narrow ArchSem interface, which can be checked to correspond by eye; both are autogenerated from the Sail model; and the Rocq proof about the concurrency model is entirely parametric on the ISA model, requiring no assumptions on it beyond it being well-typed. There is the possibility of discrepancies, but the risk of those hiding an error in the architecture that would falsify the intended property seems relatively small.

We combine the two mechanised results with a relatively small and straightforward body of paper maths. At the time of writing this relies on some additional assumptions about the ISA – these are relatively minor and should similarly be provable with our Isabelle infrastructure. The supplementary material to this paper [45] details these assumptions and contains all our proofs.

6.1 Theorem Statement

The theorem establishes that a user-level model, *UM*, is a sound abstraction of an underlying virtual-memory system model, *VM*. The system model is simply the combination of the sail-arm Arm9.4-A ISA model with our version of the VM axiomatic model of [63].

To define a user-level architecture model, we need a user-level ISA model that does memory accesses directly on virtual addresses, with no address translation. One could define that by editing the model source (replacing `AArch64.TranslateAddress` with an identity function), but that would require having two entire ISA models in the Isabelle session, which is a lot of duplication and would significantly complicate the proof. Therefore, we chose an alternative technique: we construct the user-mode ISA by writing a wrapper around the VM ISA model to erase system details, including translation table walks and translation register accesses. We have defined in Isabelle a list of *translation registers* affecting the translation process (such as `TTBR_EL1` or `TCR_EL1`), and proved it to be complete. Then the wrapper can internally handle translation table walks and translation register accesses by fulfilling those outcomes with a given translation partial state, re-exposing user-level outcomes such as general purpose register accesses and explicit memory reads and writes, but using virtual addresses. Thus, for a given translation partial state, we can define the entire user-level model, *UM*, as the combination of that constructed UM ISA model and our existing Arm user-mode axiomatic model.

To relate the behaviour of the VM and UM models, we want to relate their consistent executions, but since they have different ISA models, these are of different shapes. We relate them by *lifting* VM executions into UM executions. This includes lifting the VM initial state into a UM initial state. The former describes memory over physical addresses while the latter is indexed by virtual addresses (it thus only contains what is accessible via virtual addresses, which will not include the pagetables themselves). These virtual and physical addresses are related by a *translation characterisation* function `ttw_charac`, defined in Isabelle, which gives a (proven correct) mathematical characterisation of the Sail-Arm ISA translation-walk function's behaviour, within the configurations we target. It takes a partial memory and register state, and returns (roughly) the partial function from virtual addresses to physical addresses denoted by the page tables and system register values of that state.

The lifting of the instruction traces of VM executions is done with a *trace erasure* function, also defined in Isabelle, which takes a VM trace and erases all translation events.

We can now state our soundness theorem:

Theorem 1 (VM Abstraction). *Given a translation partial state, comprising translation registers and page table memory, such that:*

- ***Configuration:*** *The translation register configuration is supported. In particular, we presume a translation configuration with 48-bit input addresses, 48-bit output addresses, 4KB pages, no stage 2 translation, and certain architectural features disabled.*
- ***Injectivity:*** *No two virtual addresses are mapped to the same physical address.*
- ***No self-mapping:*** *The page tables do not map themselves.*

yielding a UM ISA model via the construction described above, and given a VM well-formed consistent execution such that:

- ***Matching translation registers:*** *All translation registers in the initial states of all threads match the given translation partial state.*
- ***Matching page tables:*** *The initial memory of the candidate contains the page tables of the given translation partial state.*
- ***EL0:*** *The execution remains at EL0 – the theorem covers execution up to the instruction before any context switch, with no exceptions and thus no page faults.*
- ***No abort:*** *There are no physical memory aborts, all memory accesses completed successfully.*
- ***No MSR:*** *No `MSR` instructions.*
- ***No CacheOp/TlbOp:*** *There are no cache clean or invalidate instructions* (`DC`, `IC`, `TLBIs`).
- ***No AT:*** *There are no `AT` explicit address translation instructions.*

Then the lifting operation succeeds to give a UM candidate execution that:

- *is well-formed;*
- *is consistent with both the UM ISA model and the UM axiomatic model; and*
- *whose final state is the lifting of the VM execution final state.*

The last three preconditions should follow from the EL0 requirement, but we have not proved that at present.

6.2 Proof Outline

To prove Theorem 1, we decompose into two phases. We define an intermediate *UMpa* model by joining the full Armv9.4-A ISA with our user-mode axiomatic model. This still describes an execution in terms of physical addresses but requires all translation reads to read from initial memory. *Phase 1* of the proof establishes soundness of the UMpa model with respect to the VM model. It is proven in Rocq, except for one paper-math lemma saying that if the page tables do not map themselves, then translation reads are from the initial memory. *Phase 2* establishes soundness of the UM model with respect to UMpa. It is much smaller and simpler than the above and is done in paper math [45]. Both proofs depend on various whole-ISA model properties that are proven in Isabelle, the main part of the proof effort.

6.3 Whole-ISA Properties

To give a flavour of the Isabelle whole-ISA properties, we describe the most important briefly.

Lemma 2 (Translation characterisation). *For any VM ISA trace satisfying the assumptions of Theorem 1, for any sub-trace `st` from a `TranslationStart` to a `TranslationEnd`, for any register and memory maps, satisfying our system register assumptions, if all reads in `st` come from those maps, and there are no writes, then applying `ttw_charac` to the VA from the `TranslationStart` gives the PA in the `TranslationEnd`.*

Lemma 3 (Translation footprint property). *For any VM ISA trace satisfying the assumptions of Theorem 1, any non-translation-walk memory access is preceded by a successful translation, with the footprint of the translation result containing the footprint of the memory access.*

Lemma 4 (Trace erasure properties). *For any VM ISA trace satisfying the assumptions of Theorem 1, the trace erasure succeeds and the produced trace is valid for the UM ISA model.*

6.4 Scale

The non-whitespace line counts for our definitions and proofs (available in the supplementary material [45]) are below, where ^{G} signifies automatically generated code. Typechecking ArchSem and checking the Rocq proof takes around 1 minute; typechecking Sail-Arm in Rocq takes 6m25s on an i7-10710U; typechecking Sail-Arm in Isabelle takes around 2 hours; and checking the Isabelle proof takes around 3 hours, on an Intel Xeon W-2145, with peak memory consumption around 60GB. The ISA models are largely prior work, see §4 for the patches we had to apply.

ArchSem framework	
arch-generic framework	2 912
common infrastructure	6 627
of which:	
Relational algebra library	780
Free monads library	802

ISA models	ASL	Sail	Rocq	Isabelle
sail-arm	316 052	433 530^{G}	831 995^{G}	1 170 615^{G}
sail-risc-v	n/a	22 954	140 086^{G}	146 177^{G}
sail-tiny-arm	n/a	798	3 968^{G}	5 362^{G}

VM abstraction proof		
Rocq	Manual VM axiomatic model proof	395
Isabelle	Handwritten characterisation	738
	Manual characterisation proofs	3 243
	Manual whole-ISA proof	3 478
	Auto-generated whole-ISA proof	50 304^{G}
	Translation erasure proofs	2 309
	Infrastructure	1 674
	Total Isabelle proofs	61 746
Paper	Top-level paper proof	550

Concurrency models	
Arm user axiomatic	187
Arm user promising	549
Arm VM axiomatic	569
RISC-V axiomatic	170
Sequential	161

7 Future Directions

7.1 Beyond Linear Instructions to Intra-Instruction Parallelism

As highlighted in §3.1, some instructions need non-linear intra-instruction ordering, including `CAS`, `CSEL`, and `SWP`, that Arm give specific non-linear shapes for, and any instruction that does multiple memory accesses. This includes instructions such as `STP` (store pair), scalable vector and scalable matrix (SVE and SME) instructions, and any instruction that does unaligned memory accesses, which are split into multiple single-copy-atomic accesses. Each might involve address translation – each with its 20+ translation-table-walk accesses – and these translations are not ordered with each other. Moreover, these accesses can give rise to memory faults, which take the whole instruction to an exception-raising path. We thus need to express the appropriate lack of ordering, or intra-instruction parallelism, for all these. It would not suffice to support just fork-join intra-instruction parallelism, as some of the above shapes are not expressible with that. We envisage expressing this by replacing the `iMon` monad above with a novel *async* monad:

```
Inductive aMon A (future : Set → Type) :=
| ARet (ret : A)
| ANext (call : outcome) (k : oret call → aMon A future)
| ASync (T : Set) (other : aMon T future) (k : future T → aMon A future)
| AWait (T : Set) (f : future T) (k : T → aMon A future)
| ASpec n (f : future (fin n)) (k : fin n → aMon A future)
```

This lets one start an asynchronous effectful computation, which can then be awaited on, either with `AWait`, to create a strong intra-instruction order `iio` edge, or with `ASpec`, to only create a weaker (speculative) `iio_spec` edge. This supports arbitrary dependency graphs. As for `iMon`, we have an algorithm that can validate whether a parallel analogue of an instruction trace matches an `aMon A`.

To use this within the ISA semantics the most straightforward approach would be to add explicit annotations, in the Sail or ASL code, where non-linear ordering is required. It is good to do this explicitly, as each is an architectural choice, but for non-vector instructions very little is needed – mostly in the generic memory access and translation walks, and for the above three instructions.

7.2 Axiomatic Partial Executions

As highlighted in §3.3, to properly handle UB in axiomatic models, one has to adapt conventional notions of axiomatic candidate execution, and axiomatic model definitions, to support intra-instruction-partial (and instruction-level partial) executions. To achieve this incrementality, we envisage reintroducing some operational-model concepts into axiomatic models – but in as minimal a form as we can, without an axiomatic event for every operational transition. We give the approach here, but do not develop concrete axiomatic models using it.

Operational models construct partial traces incrementally: to add a new committed event to the partial trace the model must justify that no unfinished po-previous instruction could issue an outcome that must be committed before the new event is committed. One can view Arm's locally ordered before `lob` and ordered-before `ob` as describing a read-satisfaction order. To lift the operational notion "A must be committed before B" into the axiomatic world, we define `tco` (thread commit order) and `gco` (global commit order) as analogues that describe the commit order: the order in which outcomes are marked as definitive/non-restartable. We proved that `gco` is acyclic on any consistent complete execution, as a sanity check that it is a sensible order.

```
Definition tco := (lob | ctrl | addr;po)⁺.
Definition gco := (tco | rfe | co)⁺. (* not fre *)
```

Using those, we can define the consistency of a partial candidate execution by requiring – in addition to the usual consistency axioms – that for any partial intra-instruction trace `t` in the candidate, for any event that instruction might do, this event would not be `tco` before any event po-later in the partial candidate execution. For example, if the potentially added outcome is a register write, a po-later register read that would be required to read from that write would then be `rrf` ⊆ `tco` after that write, therefore the partial execution would not be consistent. Exposing a bound on the events that a partial instruction instance might still do will require additional support in the interface. The definition leaves the semantics of complete candidates unchanged.

Acknowledgments

We thank Richard Grisenthwaite (Arm Chief Architect and EVP) for detailed discussions about the Arm architecture, and for his support. We thank Ben Laurie and Sarah de Haas (Google) for their support. This work was funded in part by Google. This work was funded in part by Arm. This work was funded in part by Intel Corporation. This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for ERC-AdG-2022, EP/Y035976/1 SAFER. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC-AdG-2017 ELVER). This work was funded in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. This work was supported in part by the Foundation of German Business (sdw) as part of its scholarship programme, funded by the German Federal Ministry of Education and Research (BMBF). This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694. This work was supported in part by UKRI3001: CHERI Research Centre. This work was supported in part by the Amazon Research Awards (ARA) program. The authors would like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Big Specification, where work on this paper was undertaken. This work was supported in part by EPSRC grant EP/Z000580/1.

References

[1] Allon Adir, Hagit Attiya, and Gil Shurek. 2003. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. *IEEE Trans. Parallel Distrib. Syst.* 14, 5 (2003), 502–515. doi:10.1109/TPDS.2003.1199067

[2] Jade Alglave. 2010. *A Shared Memory Poetics.* Ph. D. Dissertation. Université Paris 7 – Denis Diderot.

[3] Jade Alglave, Patrick Cousot, and Luc Maranget. 2016. Syntax and semantics of the weak consistency model specification language cat. *CoRR* abs/1608.07531 (2016). arXiv:1608.07531 http://arxiv.org/abs/1608.07531

[4] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. doi:10.1145/3458926

[5] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proc. DAMP 2009.*

[6] Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, and Nikos Nikoleris. 2024. Puss In Boots: on formalising Arm's Virtual Memory System Architecture (extended version). (May 2024). https://inria.hal.science/hal-04567296 working paper or preprint.

[7] Jade Alglave and Luc Maranget. 2019. The `diy7` tool. http://diy.inria.fr/. Accessed 2021-07-01.

[8] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Proc. CAV.*

[9] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: running tests against hardware. In *Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems* (Saarbruecken, Germany). Springer-Verlag, Berlin, Heidelberg, 41–44. http://dl.acm.org/citation.cfm?id=1987389.1987395

[10] 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 (2014), 7:1–7:74. doi:10.1145/2627752

[11] Saar Amar, David Chisnall, Tony Chen, Nathaniel Filardo Wesley, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture* (Toronto, Canada). Association for Computing Machinery. doi:10.1145/3613424.3614266

[12] Arm Architecture Technology Group. 2025. ASL Reference. DDI0626, version 4.0 (00bet0) https://developer.arm.com/documentation/ddi0626/latest/, accessed 2025-07-09.

[13] Arm Limited. 2025. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/. ARM DDI 0487 L.b. 14734pp.

[14] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Jon French, Kathryn E. Gray, Stephen Kell, Gabriel Kerneis, Neel Krishnaswami, Prashanth Mundkur, Robert Norton-Wright, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. 2013–2023. The Sail Instruction-Set Architecture (ISA) specification language.

[15] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages.* doi:10.1145/3290384 Proc. ACM Program. Lang. 3, POPL, Article 71.

[16] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *In Proc. 33rd International Conference on Computer-Aided Verification.* Extended version available at https://www.cl.cam.ac.uk/~pes20/isla/isla-cav2021-extended.pdf.

[17] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. 2022. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture. In *Proceedings of the 31st European Symposium on Programming*. Springer, 174–203. doi:10.1007/978-3-030-99336-8_7

[18] Thomas Bauereiss and Thomas Sewell. 2022–2025. Sail Armv9.4-A ISA properties in Isabelle. https://github.com/rems-project/archsem-vm-isa-proofs

[19] Thomas Bauereiss, Patrick Taylor, Alasdair Armstrong, and Peter Sewell. 2023. ACL2-to-Sail translator and the resulting Sail x86 ISA model. https://github.com/rems-project/sail-x86-from-acl2.

[20] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 1770–1800. doi:10.1145/3571254

[21] William W. Collier. 1992. *Reasoning about parallel architectures.* Prentice Hall.

[22] Zhenyang Dai, Shuang Liu, Vilhelm Sjöberg, Xupeng Li, Yu Chen, Wenhao Wang, Yuekai Jia, Sean Noble Anderson, Laila Elbeheiry, Shubham Sondhi, Yu Zhang, Zhaozhong Ni, Shoumeng Yan, Ronghui Gu, and Zhengyu He. 2024. Verifying Rust Implementation of Page Tables in a Software Enclave Hypervisor. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 1218–1232. doi:10.1145/3620665.3640398

[23] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Rosu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1133–1148. doi:10.1145/3314221.3314601

[24] Shaked Flur, Jon French, Kathryn Gray, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2017. rmem. www.cl.cam.ac.uk/~pes20/rmem/.

[25] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA)*. 608–621. doi:10.1145/2837614.2837615

[26] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

[27] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. In *The 44st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*. 429–442. doi:10.1145/3009837.3009839

[28] Kourosh Gharachorloo. 1995. *Memory Consistency Models for Shared-Memory Multiprocessors.* Ph. D. Dissertation. Stanford University.

[29] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*, Jean-Loup Baer, Larry Snyder, and James R. Goodman (Eds.). ACM, 15–26. doi:10.1145/325164.325102

[30] Shilpi Goel, Warren A. Hunt Jr., and Matt Kaufmann. 2017. Engineering a Formal, Executable x86 ISA Simulator for Software Verification. In *Provably Correct Systems*. 173–209. doi:10.1007/978-3-319-48628-4_8

[31] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture (Waikiki)*. 635–646. doi:10.1145/2830772.2830775

[32] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[33] Angus Hammond[1], Zongyuan Liu[1], Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. 2024. An axiomatic basis for computer programming on the relaxed Arm-A architecture: the AxSL logic. In *Proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages*. doi:10.1145/3632863 [1]These authors contributed equally. Proc. ACM Program. Lang. 8, POPL, Article 21.

[34] Sander Huyghebaert, Steven Keuchel, Coen De Roover, and Dominique Devriese. 2023. Formalizing, Verifying and Applying ISA Security Guarantees as Universal Contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng,

Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2083–2097. doi:10.1145/3576915.3616602

[35] Intel. 2002. A Formal Specification of Intel Itanium Processor Family Memory Ordering. http://download.intel.com/design/Itanium/Downloads/25142901.pdf.

[36] Jieung Kim, Ronghui Gu, and Zhong Shao. 2024. SimplMM: A simplified and abstract multicore hardware model for large scale system software formal verification. *J. Syst. Archit.* 147 (2024), 103049. doi:10.1016/J.SYSARC.2023.103049

[37] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70. doi:10.1145/2560537

[38] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. doi:10.1145/2535838.2535841

[39] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf

[40] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 839–856. doi:10.1109/SP40001.2021.00049

[41] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and Verification of the Arm Confidential Compute Architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 465–484. https://www.usenix.org/conference/osdi22/presentation/li

[42] Gregory Malecha, Gordon Stewart, Frantisek Farka, Jasper Haag, and Yoichi Hirai. 2022. Developing With Formal Methods at BedRock Systems, Inc. *IEEE Secur. Priv.* 20, 3 (2022), 33–42. doi:10.1109/MSEC.2022.3158196

[43] Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 257–275.

[44] Prashanth Mundkur, Rishiyur S. Nikhil, Jon French, Brian Campbell, Robert Norton-Wright, Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher Pulte, Peter Sewell, Alexander Richardson, Hesham Almatary, Jessica Clarke, Nathaniel Wesley Filardo, Peter Rugg, and Scott Johnson. 2014–2023. Sail RISC-V instruction-set architecture (ISA) model.

[45] Thibaut Pérami, Thomas Bauereiss, Brian Campbell, Zongyuan Liu, Nils Lauermann, Alasdair Armstrong, and Peter Sewell. 2026. Supplementary material for ArchSem: Reusable Rigorous Semantics of Relaxed Architectures. doi:10.17863/CAM.123654 Cambridge Apollo data repository.

[46] Thibaut Pérami, Yeji Han, Zongyuan Liu, Nils Lauermann, Jean Pichon-Pharabod, Brian Campbell, Alasdair Armstrong, Ben Simner, and Peter Sewell. 2022–2025. ArchSem. https://github.com/rems-project/archsem

[47] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan. 2019), 31 pages. doi:10.1145/3290382

[48] Anton Podkopaev, Egor Namakonov, Ori Lahav, and Ilya Kaysin. 2019. Intermediate Memory Model (IMM) and compilation correctness proofs for it. https://github.com/weakmemory/imm

[49] Christopher Pulte. 2018. *The Semantics of Multicopy Atomic ARMv8 and RISC-V.* Ph. D. Dissertation. University of Cambridge. https://www.repository.cam.ac.uk/handle/1810/292229.

[50] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. doi:10.1145/3158107

[51] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1–15. doi:10.1145/3314221.3314624

[52] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022. Extending Intel-x86 consistency and persistency: formalising the semantics of Intel-x86 memory types and non-temporal stores. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. doi:10.1145/3498683

[53] Alastair Reid. 2016. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In *FMCAD 2016* (Mountain View, CA, USA). 161–168. https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf

[54] Alastair Reid. 2022. Towards a formal x86 ISA specification. Slides presented at the workshop on Novel Architecture and Novel Design Automation (NANDA), Imperial College http://cc.doc.ic.ac.uk/nanda/, https://alastairreid.github.io/talks/towards-a-formal-x86-specification-ImperialCollege-2022-09-05.pdf (accessed 2025-07-09).

[55] Hadrien Renaud. 2024. Executable semantics of Arm's Architecture Specification Language. In *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*. Saint-Jacut-de-la-Mer, France. https://hal.science/hal-04406399

[56] Ian Roessle, Freek Verbeek, and Binoy Ravindran. 2019. Formally verified big step semantics out of x86-64 binaries. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 181–195. doi:10.1145/3293880.3294102

[57] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 825–840. doi:10.1145/3519939.3523434

[58] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of PLDI 2012, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*. 311–322. doi:10.1145/2254064.2254102

[59] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*. 175–186. doi:10.1145/1993498.1993520

[60] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave. 2009. The Semantics of x86-CC Multiprocessor Machine Code. In *Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 379–391. doi:10.1145/1594834.1480929

[61] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. (Research Highlights).

[62] Ben Simner, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell. 2025. Precise exceptions in relaxed architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture, ISCA 2025, Tokyo, Japan, June 21-25, 2025*. ACM, 211–224. doi:10.1145/3695053.3731102 Best paper award.

[63] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *Proceedings of ESOP 2022: 31st European Symposium on Programming, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany*.

[64] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. 2020. ARMv8-A system semantics: instruction fetch in relaxed architectures (extended version). In *Proceedings of the 29th European Symposium on Programming*.

[65] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. 1991. Formal Specification of Memory Models. In *Scalable Shared Memory Multiprocessors*. Kluwer, 25–42.

[66] Viktor Vafeiadis. 2021. Arm8 weak memory model formalisation in Coq. https://github.com/vafeiadis/arm-model

[67] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. Document Version 20191213. Contributors: Arvind, Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, Ken Dockser, Roger Espasa, Shaked Flur, Stefan Freudenberger, Marc Gauthier, Andy Glew, Jan Gray, Michael Hamburg, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Alexandre Joannou, Olof Johansson, Ben Keller, David Kruckemyer, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Margaret Martonosi, Joseph Myers, Vijayanand Nagarajan, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Christopher Pulte, Jose Renau, Josh Scheid, Colin Schmidt, Peter Sewell, Susmit Sarkar, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Andrew Waterman, Robert Watson, Derek Williams, Andrew Wright, Reinoud Zandijk, and Sizhuo Zhang.

[68] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. doi:10.48456/tr-987

[69] Wilkes, Wheeler, and Gill. 1956. Introduction to Programming for the EDSAC. University Mathematical Laboratory, Cambridge.

[70] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. doi:10.1145/3371119

Received 2025-07-10; accepted 2025-11-06