An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic

ANGUS HAMMOND[*], University of Cambridge, UK
ZONGYUAN LIU[*], Aarhus University, Denmark
THIBAUT PÉRAMI, University of Cambridge, UK
PETER SEWELL, University of Cambridge, UK
LARS BIRKEDAL, Aarhus University, Denmark
JEAN PICHON-PHARABOD, Aarhus University, Denmark

Very relaxed concurrency memory models, like those of the Arm-A, RISC-V, and IBM Power hardware architectures, underpin much of computing but break a fundamental intuition about programs, namely that syntactic program order and the reads-from relation always both induce order in the execution. Instead, out-of-order execution is allowed except where prevented by certain pairwise dependencies, barriers, or other synchronisation. This means that there is no notion of the 'current' state of the program, making it challenging to design (and prove sound) syntax-directed, modular reasoning methods like Hoare logics, as usable resources cannot implicitly flow from one program point to the next.

We present AxSL, a separation logic for the relaxed memory model of Arm-A, that captures the fine-grained reasoning underpinning the low-overhead synchronisation mechanisms used by high-performance systems code. In particular, AxSL allows transferring arbitrary resources using relaxed reads and writes when they induce inter-thread ordering. We mechanise AxSL in the Iris separation logic framework, illustrate it on key examples, and prove it sound with respect to the axiomatic memory model of Arm-A. Our approach is largely generic in the axiomatic model and in the instruction-set semantics, offering a potential way forward for compositional reasoning for other similar models, and for the combination of production concurrency models and full-scale ISAs.

CCS Concepts: • **Theory of computation** → **Separation logic**; • **Computer systems organization** → *Multicore architectures.*

Additional Key Words and Phrases: relaxed memory models, program logic, separation logic, Arm, Iris

ACM Reference Format:
Angus Hammond, Zongyuan Liu, 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. https://doi.org/10.1145/3632863

[*] These authors contributed equally to this research.

Authors' addresses: Angus Hammond, Angus.Hammond@cl.cam.ac.uk, University of Cambridge, Computer Laboratory, JJ Thomson Avenue, Cambridge, UK, CB3 0FD; Zongyuan Liu, zy.liu@cs.au.dk, Aarhus University, Åbogade 34, Aarhus, Denmark, 8200; Thibaut Pérami, Thibaut.Perami@cl.cam.ac.uk, University of Cambridge, Computer Laboratory, JJ Thomson Avenue, Cambridge, UK, CB3 0FD; Peter Sewell, Peter.Sewell@cl.cam.ac.uk, University of Cambridge, Computer Laboratory, JJ Thomson Avenue, Cambridge, UK, CB3 0FD; Lars Birkedal, birkedal@cs.au.dk, Aarhus University, Åbogade 34, Aarhus, Denmark, 8200; Jean Pichon-Pharabod, jean.pichon@cs.au.dk, Aarhus University, Åbogade 34, Aarhus, Denmark, 8200.

1 INTRODUCTION

Systems code, such as operating system and hypervisor kernel code, is a prime target for software verification, being security-critical yet relatively small. However, it is highly concurrent, which raises two questions: What model to verify it above? And what verification theory to use? The Arm-A architecture is used in essentially all mobile devices, and its base ("user") relaxed concurrency model is now reasonably well-understood and stable [Arm Ltd. 2023, Ch.B2],[Alglave et al. 2021, 2014; Deacon 2016; Flur et al. 2017; Pulte et al. 2018]. However, there is little program verification theory or tooling that applies directly to Arm-A.

In this paper, we develop a separation logic for Arm-A "user" concurrency that is expressive, supporting local reasoning with higher-order ghost state and invariants, and mechanised in Coq, using the Iris program logic framework [Jung et al. 2018, 2015]. The Arm-A concurrency model is particularly challenging for program-logic reasoning because it (like RISC-V and IBM Power, but unlike x86) permits load-store reordering, as in the classic "load buffering" LB shape of Fig. 1. This means that the union of program order (po) and the reads-from relation (rf) is not guaranteed to be acyclic — but for compositional reasoning, one wants to attach assertions to particular program points, and program logics usually rely on the strength of program order captured by that acyclicity; they let resources implicitly flow in the proof context from one program point to the next. Previous program logics have either assumed po ∪ rf acyclic (which requires extra barriers), e.g. FSL++ [Doko 2021; Doko and Vafeiadis 2017], GPS [Turon et al. 2014], and iRC11 [Dang et al. 2020], or lack ghost state, e.g. FSL [Doko and Vafeiadis 2016], which makes the logic substantially less expressive and more awkward to use, or give extremely weak guarantees for non-synchronised reads, e.g. RSL [Vafeiadis and Narayan 2013]. The Lace logic [Bornat et al. 2015] targeted relaxed architectural models but lacked a proof of soundness, and the Ogre and Pythia logic [Alglave and Cousot 2017] is a refinement of Owicki-Gries [Owicki and Gries 1976] that is parameterised by (and sound for) a range of relaxed models, but (like Owicki-Gries) lacks thread-local modular reasoning.

a: r_1 := ldr [x]	c: r_2 := ldr [y]
b: str [y] 1	d: str [x] 1

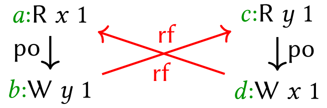

Fig. 1. LB+pos

In contrast to those previous program logics, to allow sound usage of ghost resources even in the presence of LB, we prevent implicit flow of usable resources between program points along po, allowing it only along the ordered-before ordering (ob) of the Arm-A memory model. In Arm-A, there are many ways to create ob ordering. We allow explicit reasoning about those if need be, by exposing the structure of the axiomatic model, letting one reason about the low-cost ordering that Arm-A guarantees from various forms of dependency (RSL, FSL, FSL++, GPS, and iGPS are all for C11 or RC11, without dependencies).

Stepping back, why would one want to reason directly above an architecture concurrency model? After all, high-level language concurrency models, e.g. C/C++11 [Batty et al. 2011; Boehm and Adve 2008] and the Linux kernel memory model, LKMM [Alglave et al. 2018; McKenney et al. 2020], were designed to obviate the need to program and reason about specific underlying architectures, with extensive work on the correctness of their compilation schemes [Batty et al. 2012; Lahav et al. 2017; Manerkar et al. 2016; Sarkar et al. 2012], and one would not envisage manual proof about large bodies of assembly code. There are three main reasons.

First, those C/C++ language-level models are fundamentally flawed for highly relaxed code because of the out-of-thin-air problem [Becker 2011, §23.9p9] [Batty et al. 2015]: they allow arbitrary values to be created, e.g. for the Fig. 2 LB+datas shape of relaxed atomic accesses and source-language data dependencies. Thin-air values are not believed to arise for conventional compilers and hardware, but it has proven challenging to define tractable semantics that exclude them while remaining sound w.r.t. conventional compiler and hardware optimisations — especially compiler dependency removal and hardware load-store reordering. The LKMM forbids thin-air outcomes by assuming some dependencies are respected, and in specific coding idioms they often are, but in general they can be removed by conventional compiler optimisations. There have been many attempts to solve this problem [Chakraborty and Vafeiadis 2019; Jeffrey and Riely 2016; Kang et al. 2017; Lee et al. 2020; Paviotti et al. 2020; Pichon-Pharabod and Sewell 2016], but so far none have been adopted — so we simply do not yet have any high-level language semantics suitable for reasoning about deployed highly relaxed code. In contrast, architecture concurrency models for Arm-A (and other architectures, including x86, RISC-V, and IBM Power) are now well-established [Alglave et al. 2021, 2014; Arm Ltd. 2023; Deacon 2016; Flur et al. 2017; Owens et al. 2009; Pulte et al. 2018; Sarkar et al. 2011, 2009; Waterman and Asanović 2019], and do not suffer from the thin-air problem: these architectures guarantee to respect certain syntactic dependencies, ruling out thin-air. These architectural models thus give us a solid foundation that we can reason above.

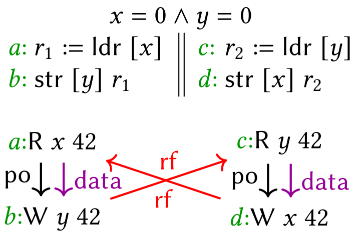

Fig. 2. LB+datas

Second, ultimately, the machine-code binary is what runs — and therefore one ultimately wants to verify down to the (concurrent) machine semantics, even if the bulk of one's source-language verification is at the C level or above. There are several possible approaches to this: for example, one might have a source language with more restricted concurrency (without relaxed accesses), and then some verified compilation result down to the machine semantics [Cho et al. 2022; Tao et al. 2021]. But production systems-code practice does use relaxed accesses for performance, and hence reasoning about them is an important problem. We thus aim here to first understand how to reason directly about the binary, where we have a good underlying model; future work can then use this as the basis for verified compilation or other verification approaches for higher-level code.

Third, systems code relies, in small but crucial parts, on assembly which is not C-language expressible — e.g. for particular barriers, and for management of systems features of the underlying architecture (instruction and data cache management [Simner et al. 2020], virtual memory [Simner et al. 2022][Arm Ltd. 2023, B2.3], exceptions, etc.). We do not cover systems semantics here, but our approach is designed to generalise to it.

Contributions. We develop a separation logic for Arm-A "user" concurrency, AxSL, that is expressive, supporting reasoning with higher-order ghost state and invariants, and mechanised in Coq, using the Iris program logic framework.

We do this specifically for the Arm-A concurrency model, and only for an idealised instruction set architecture (ISA), but our approach is designed to generalise in both respects. For the ISA, our idealised ISA semantics and base logic are defined above the microinstructions of the Sail "outcome" interface [Gray et al. 2015][Pulte et al. 2018, §6.1][Pulte 2018, §2.3], so they should generalise straightforwardly to the full ISA of Arm-A or RISC-V. For the concurrency model, our approach is largely generic in the structure of the Arm-A axiomatic model, so this work offers a path towards similar logics for other architecture axiomatic models (e.g. the rather similar RISC-V "user" model), or, more speculatively, to extensions covering Arm-A systems semantics [Simner et al. 2022, 2020]. Moreover, the Arm-A architecture reference manual specifies its concurrency architecture in this axiomatic style [Arm Ltd. 2023, Ch.B2] and is actively maintained and occasionally changed, so (while semantics for new features have been developed both within and outside Arm, and in multiple styles), it is desirable to be able to track the reference-manual version with minimal effort.

We describe the program-logic and relaxed-memory context in §2. We explain the key ideas of our logic informally in §3, and in §4 we describe the language we consider – the combination of a simplified assembly language and the real LB-permitting Arm-A axiomatic concurrency model – and how we give its semantics in a way that makes it possible to build an expressive logic featuring higher-order ghost state. In §5, we describe the rules of our logic and exercise them on small, representative examples. In §6, we show how we embed the axiomatic memory model of Arm-A in Iris; we present our non-standard definition of weakest precondition and non-standard proof of adequacy for an axiomatic memory model in Iris. We discuss related work in §7, and how our work can be used and extended further in §8. The logic, its soundness, and the examples are formalised in Coq using the Iris separation logic framework; the full development is available [Hammond et al. 2023].

Limitations. To avoid adding overwhelming complexity to an already complex topic, we only consider the "user" Arm-A memory model of 2018 [Pulte et al. 2018], and not more recent extensions and changes: no mixed-size accesses, no instruction fetching, no virtual memory, and no pick dependencies, although these extensions are all in the shape that our approach supports. Also for simplicity, we use an idealised assembly language, but, as mentioned, give its semantics in terms of the Sail interface monad that supports the full Arm ISA. Our logic is sound, but we did not investigate its completeness. Our logic provides support and abstraction when reasoning about synchronisation (through ob), but not when reasoning about coherence (per-location SC).

2 CONTEXT: PROGRAM LOGICS AND RELAXED CONCURRENCY

Early work on program verification, in a sequential setting, could assume the existence of a simple program state of memory values, updated by each instruction, and program proof could be done by annotating a flowchart (as per Turing [Morris and Jones 1984; Turing 1949] and Floyd [Floyd 1967]), or syntactic program points (as per Naur [Naur 1966] and Hoare [Hoare 1969]), with assertions on that state. In this setting, a fact about a part of the state untouched by some instruction remains true (and usable for program proof) after the instruction, though managing such framing had to be done manually. The first separation logics, of Reynolds, O'Hearn, and Yang [O'Hearn et al. 2001; Reynolds 2002], refined this view with a separating conjunction, allowing assertions to express ownership of some part of such a state, with an explicit frame rule. Simple concurrent separation logics, e.g. CSL [Brookes and O'Hearn 2016], are broadly similar except that ownership of parts of the state can be transferred at lock acquire and release points; facts about owned parts of the state remain true from one program point to the next, except where the state they mention is explicitly modified by the intervening instruction.

In a relaxed-memory concurrent setting, however, there is no simple notion of program state, acted on by all threads in some global interleaving: threads do not execute in-order, and different threads can observe events in incompatible orders. To capture this, the underlying semantics have quite different forms to classical sequential or sequentially consistent concurrent semantics. Two styles of semantics for architectural relaxed-memory concurrency are common: abstract-microarchitectural operational models explain how the allowed observable behaviour arises from explicit speculative execution and event propagation, but with roll-back when speculation turns out to violate some constraint, e.g. [Higham et al. 2007; Owens et al. 2009; Pulte et al. 2018; Sarkar et al. 2011], while axiomatic models define the allowed observable behaviour more concisely as predicates on candidate complete execution graphs, e.g. [Alglave et al. 2010, 2014; Gharachorloo 1995; Kohli et al. 1993], but do not straightforwardly support the incremental construction of valid executions. A third, "Promising", style is, very roughly, intermediate between the two [Pulte et al. 2019]. All are challenging to work with, in different ways, as we discuss in §3.2.

```
(* Coherence-after *)
let ca = fr | co
(* Observed-by *)
let obs = rfe | fre | coe
(* Dependency-ordered-before *)
let dob = addr | data
| ctrl; [W]
| (ctrl | (addr; po)); [ISB]; po; [R]
| addr; po; [W]
| (ctrl | data); coi
| (addr | data); rfi
(* Atomic-ordered-before *)
let aob = rmw
| [range(rmw)]; rfi; [A | Q]
(* Barrier-ordered-before *)
let bob = po; [dmb.full]; po | [L]; po; [A]
| [R]; po; [dmb.ld]; po
| [A | Q]; po
| [W]; po; [dmb.st]; po; [W] | po; [L]
| po; [L]; coi
(* Locally ordered-before *)
let lob = dob | aob | bob
(* Ordered-before *)
let ob = (obs | lob)+
(* Internal visibility requirement *)
acyclic po-loc | ca | rf
(* External visibility requirement *)
irreflexive ob
(* Atomicity requirement *)
empty rmw & (fre; coe) as atomic
```

Fig. 4. Arm-A axiomatic model by Deacon [Pulte et al. 2018] (with lob separated out, following later Arm models [Alglave et al. 2021]), in herd's cat syntax [Alglave et al. 2014] for relational algebra. Here `|`, `&`, `;`, and `+` are relational union, intersection, composition, and transitive closure; `[W]`, `[R]`, `[L]`, `[A]` and `[Q]` are the identity relations over all write, read, release, acquire and acquirePC events; `[ISB]`, `[dmb.full]`, `[dmb.st]`, `[dmb.ld]` are the identity on those barrier events; `addr`, `data`, and `ctrl` are the syntactic dependency-relation subsets of program order `po`; `po-loc` relates same-address memory accesses in `po`; `co` is coherence over writes; the derived `fr` relates reads to coherence successors of the write they read from; `rmw` is the successful read/write-exclusive pairs; and the `rf`, `co`, and `fr` relations are subdivided into their "internal" (same-thread) and "external" (different-thread) parts, suffixed `i` and `e` respectively. The main "axiom" requires that ordered-before (`ob`) is irreflexive.

We base the current work on an axiomatic model. In these, a program gives rise to a large set of candidate complete execution graphs X, each with a function X.lab from event IDs to events, a program order relation over event IDs (po) within each thread, with reads of arbitrary values and writes of values consistent with the instruction semantics, and with arbitrary reads-from (rf) and coherence (co) relations between them. An axiomatic concurrency model typically defines various relations derived from these, e.g. for Arm-A the Fig. 4 model defines an *observed before* (ob) relation, and imposes constraints on those, in particular that ob is acyclic. The semantics of a program is the set of all candidate complete execution graphs that satisfy those properties and are consistent with the intra-instruction semantics. For example, the candidate execution for LB+pos in Fig. 1 is allowed by the Arm-A axiomatic model because the plain po relation, between reads and writes to different addresses, is not included in the ordered-before ob that is required to be acyclic (or in the internal or atomicity requirements). The candidate execution for LB+data at the bottom of Fig. 2 is forbidden in Arm-A because the intra-thread syntactic data dependencies create data edges, which are included in the Arm-A *locally ordered before* lob relation, and that and the inter-thread reads-from relation rfe are both contained in ob. For contrast the candidate execution for LB+data in which a reads from the initial state, above, is allowed.

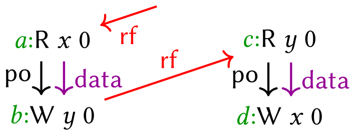

Fig. 3. An SC execution of LB+datas

For some, relatively simple, forms of relaxed concurrency, one can adapt separation logic relatively straightforwardly. For example, rely/guarantee reasoning with acquire/release reads and writes lets one do thread-modular proofs, in which a thread might gain some resource at an acquire read, manipulate it freely, and then pass it on with a release write – with the resource still persisting from one program point to the next between those points (except where explicitly modified by this thread), as a read-acquire is ordered with all po-successors and a write-release with all po-predecessors.

The effect of reading from a shared variable on the thread's logical state is accounted for thread-locally by relying on a protocol or invariant to abstract the possible actions of other threads. The protocol constrains what logical resources are transferred when accessing shared variables. In a candidate execution, one can see this as annotating the incoming (to reads) and outgoing (from writes) reads-from edges, for the part of the graph for each thread, with the resources that get transferred along them (Fig. 5).

In this view, as described for RSL, the events of the execution graph act following *flow implications*: "the annotation is locally valid around that action [when] basically the sum of the annotated heaps on the incoming edges should equal the sum of the annotated heaps on the outgoing edges, modulo the effect of [the] action".

FSL generalises RSL to reason about C11's release and acquire fences, but its assertions are still persistently freely usable along po, so they have to choose between soundness in the presence of load buffering (FSL) and support for ghost state, at the cost of requiring po ∪ rf acyclic (FSL++). We describe these and related logics in more detail in §7.

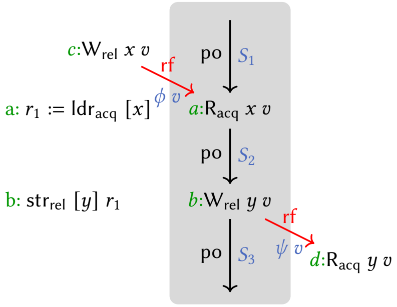
c:W$_{\mathsf{rel}}$ x v
rf
po
S_1
a: r_1 := ldr$_{\mathsf{acq}}$ [x]
ϕ v
a:R$_{\mathsf{acq}}$ x v
po
S_2
b: str$_{\mathsf{rel}}$ [y] r_1
b:W$_{\mathsf{rel}}$ y v
rf
po
S_3
ψ v
d:R$_{\mathsf{acq}}$ y v

Fig. 5. Thread-local part (in grey) of a candidate execution, annotated with the logical resources in blue flowing on edges. The thread program is on the left. Resources $S_{1,2,3}$ are those in hand at each program point, and the protocol specifies the resources ϕ v and ψ v passed along the release-acquire edges for x and y.

3 KEY IDEAS

3.1 The First Problem: Relaxed Thread-local Ordering

The biggest challenge for reasoning about the more relaxed behaviour of mainstream (non-TSO) relaxed architectures, including Arm-A, RISC-V, and IBM Power, arises from the fact that they all permit out-of-order execution of program-ordered loads and stores, except where there is some dependency or barrier. This means that a resource gained on a load *cannot* be deemed to implicitly persist through to any program-order-later store where it might be passed on. For example, consider a thread consisting of two interleaved copies of the left thread of LB+datas (operating on disjoint addresses), as in Fig. 6. The data dependencies order a with c, and b with d, but that is all the ordering we get. In particular, nothing orders the last read b, before the first write c – in contrast to the release/acquire case.

The Arm-A axiomatic model's locally-ordered-before (lob) relation specifies what thread-local ordering is respected, as introduced by barriers, synchronising accesses (store release, acquire reads, etc.), and register-to-register dependencies. All but the strongest barriers and synchronising accesses impose only a partial ordering and allow some intra-thread concurrency. In particular, some register dependencies merely impose a pairwise ordering of events; as such, they are particularly cheap, and are one of the motivations to directly write assembly for high-performance code, for example in the Linux kernel's pervasive RCU library.

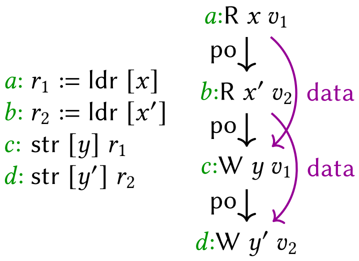

Fig. 6. Intra-thread concurrency

Our first key idea is that by attaching resources only to locally-ordered-before edges, rather than all of program order, we can make a sound logic even for relaxed architectures exhibiting load buffering and intra-thread concurrency. However, for practical and compositional reasoning, we want to annotate a program text, not the large set of its candidate executions. Moreover, to identify when ordering will arise from register dependencies to program-order-later events, it suffices to keep track of the *source* of each register value, not just its value. Concretely, a load a into register r

of a value v, from some location following a protocol ϕ, will give us

$$(r \overset{\mathrm{r}}{\mapsto} v@\{a\}) * (a \looparrowright (\phi\ v))$$

Here our *register points-to* assertion $r \overset{\mathrm{r}}{\mapsto} v@E$ keeps track of the set E of (thread-local) events that it stems from, along with r's current value v, while $a \looparrowright (\phi\ v)$ records the resources gained (according to protocol ϕ) from the load, *tying* them to its event ID a. (The $\phi, \psi, \ldots$ are per-location value-based protocols, which we later generalise.)

These register points-to and tied resources then do flow down to later program points (except where transferred away), but, crucially, they can only be used for an event b when a is locally-ordered-before b, e.g. where b is a program-order and data-dependent write after a, which might consume some or all of the resources in passing them to another thread. It is tempting to try to combine these two assertions into one, bypassing the indirection, as $r \overset{\mathrm{r}}{\mapsto} v\&(\phi\ v)$, but this breaks down for all but the simplest use cases: moving the contents of a register into another must distribute the resources, or use indirection as we do via events. Lace logic [Bornat et al. 2015] had a somewhat similar mechanism, but more explicitly in terms of edges than sources, which fits their setting where they dictate ordering (à la Crary and Sullivan [Crary and Sullivan 2015]) better, but is less convenient for ours, where ordering emerges in program order. In general, of course, there may be many dynamic instances – and hence memory events – arising from each static instruction; that can also be dealt with within the logic, by existential quantification and counters for event IDs [Alglave and Cousot 2017; Lamport 1977].

Crucially, we allow any Iris proposition to be tied to an event. This includes any piece of ghost state gs, embedded into an Iris proposition as $\boxed{gs}$. Ghost state is very flexible [Dinsdale-Young et al. 2013, 2010; Jung et al. 2018, 2015; Svendsen and Birkedal 2014], and, as usual in Iris, we use it both (1) to track the physical state (by enforcing in the definition of weakest precondition that it keeps in sync with physical state introduced in §§3.2 and 4.3), but piecemeal, so that we for example can talk about the state of a single register; and (2) to track the logical state of a program, for example with an exclusive permission to commit to a value, where owning such a permission to commit refutes observing another thread having done something that required having committed to a value (as in §5.3). In a sense, ghost state instruments the physical state of the operational semantics, but unlike physical state, ghost state can be updated freely by a *view shift* $P \Rrightarrow Q$, as long as the update is frame-preserving, meaning that it does not contradict other pieces of ghost state; the view shift can be viewed as a generalised implication. Finally, our $a \looparrowright P$ assertion is itself defined using ghost state, using the fact that Iris ghost state is higher-order, in the sense that it is mutually defined with Iris propositions.

Using these assertions, we can write concrete proofs for synchronisation involving thread-local dependencies, as sketched in Fig. 7, without relying on the program-order strength of release-acquire reasoning we illustrated in Fig. 5. (This proof sketch is more complicated than needed for LB+datas, which has a very simple proof just asserting all writes write 0, but it generalises to variations of LB, as we show in §5.) The initial logical state of the thread on line 1, S_1, includes a register points-to for r_1 containing unknown, irrelevant data, which we write with an underscore: $r_1 \overset{\mathrm{r}}{\mapsto} \_$, and some potential extra logical state R_0. The load a reads some value v_1, so now we have $r_1 \overset{\mathrm{r}}{\mapsto} v_1@\{a\}$ and $a \looparrowright (\phi\ v_1)$. When performing the store b on line 4, the proof rule requires us to establish the corresponding flow implication.

```
1  $\{r_1 \overset{\mathrm{r}}{\mapsto} \_ * R_0\}$ // $S_1$
2  $a$: $r_1$ := ldr [$x$]
3  $\{\exists v_1.\, r_1 \overset{\mathrm{r}}{\mapsto} v_1@\{a\} * (a \looparrowright (\phi\ v_1)) * R_0\}$ // $S_2$
4  $b$: str [$y$] $r_1$    $(R_0 * \phi\ v_1) \Rrightarrow (\psi\ v_1)$
5  $\{r_1 \overset{\mathrm{r}}{\mapsto} v_1@\{a\}\}$ // $S_3$
```

Fig. 7. Proof sketch for a plain-access (non-release/acquire) version of Fig. 5. The flow implication is on line 4.

Because the store has a data dependency on a, we get to use not only the ambient R_0, but also the $\phi\ v_1$ tied to a, to establish (because this is a store) the protocol for v_1 for y, $\psi\ v_1$. The flow implication for b that the proof rule requires us to establish is thus $(R_0 * \phi\ v_1) \Rrightarrow (\psi\ v_1)$.

If the data dependency between a and b is removed (so the store, e.g. now of a constant, can execute early, and hence the relaxed LB behaviour of Fig. 1, where both reads read a non-zero value, is allowed by Arm-A), then the proof does not go through anymore, as desired, because the flow implication for b no longer has $\phi\ v_1$ available. This illustrates how our assertions allow us to soundly use ghost state to reason about relaxed architectures exhibiting load buffering and intra-thread concurrency.

Framing. We are separating resources flowing from different sources to different targets by necessity. Relatedly, one of the points of separation logic is allow separate resources to flow side-by-side, for convenience (specifically, for modularity). In the example of Fig. 6, reasoning about c is not allowed to use the resource from b, only from a — thanks to framing, it does not need to mention the resource from b either (similarly, reasoning about d is not allowed to use the resource from a, and does not need to mention them either). In addition to framing a tied resource off, we also support splitting tied resources, so that given an instruction that merely needs $a \looparrowright P$, we can split $a \looparrowright (P * Q)$ into $(a \looparrowright P) * (a \looparrowright Q)$ and frame the latter off, as in Fig. 8, see §6.1.

```
$a$: $r$ := ldr [$x$]
$\{r \overset{\mathrm{r}}{\mapsto} v@\{a\} * a \looparrowright (P * Q)\}$
$\{r \overset{\mathrm{r}}{\mapsto} v@\{a\} * (a \looparrowright P) * (a \looparrowright Q)\}$
   $\{r \overset{\mathrm{r}}{\mapsto} v@\{a\} * a \looparrowright P\}$
$b$: str [$y$] $r$ // uses $P$
   $\{r \overset{\mathrm{r}}{\mapsto} v@\{a\} * a \looparrowright \top\}$
   $\{r \overset{\mathrm{r}}{\mapsto} v@\{a\}\}$
$\{r \overset{\mathrm{r}}{\mapsto} v@\{a\} * a \looparrowright Q\}$
$c$: str [$z$] $r$ // uses $Q$
$\{r \overset{\mathrm{r}}{\mapsto} v@\{a\} * a \looparrowright \top\}$
```

Fig. 8. Splitting tied resources

3.2 The Second Problem: Operationalising the Relaxed Arm-A Model

The next challenge is that of selecting – or developing – a version of the Arm-A concurrency architecture to underlie the soundness proof for our logic. A priori, one might use existing abstract-microarchitectural operational [Pulte et al. 2018], axiomatic [Pulte et al. 2018], or Promising-Arm [Pulte et al. 2019] models, which are proved equivalent (for the features covered by all). We would like to express the logic as an instantiation of Iris [Jung et al. 2018, 2015], an expressive separation logic framework, to get the benefits of its higher-order ghost state, guarded recursion, and existing mechanisation. That requires the underlying semantics to be phrased as a small-step operational semantics, for the logical setup for higher-order ghost state to apply, and it should work ‘enough’ along program order for the soundness proof of our syntax-directed proof rules to be tractable.

The abstract microarchitectural operational model is explanatory, based on hardware intuition, and it is operational, but it is in this respect too close to hardware, with explicit out-of-order execution; it also splits memory reads and writes into multiple fine-grained events. The axiomatic model as normally presented is not straightforwardly operational: phrased as acyclicity requirements on the ob and certain other derived relations of a whole-program complete candidate execution, expressed using relational algebra with fixpoints over basic relations po, data, rf, etc. One might imagine constructing an operational model from the axiomatic model by fiat, with a state that is a set of events, and steps that add an arbitrary event and recheck the axiomatic-model validity predicate, but for Arm-A (and similarly RISC-V and IBM Power), because po ∪ rf is not acyclic, this cannot straightforwardly follow program order, as reads would have to sometimes read from events that have not yet been introduced. One might follow ob, but that would be at odds with the structure of the soundness proof. Or one might permit such reads to read new symbolic values, and propagate those through the instruction semantics, but that adds substantial complexity.

Instead, we develop a novel operationalisation of the axiomatic memory model in a mixed operational-axiomatic style (§4), our **second key idea**. This *opax* semantics is sufficiently close to a small-step operational semantics that it is not too difficult to instantiate the Iris logical framework to it, it works enough along program order for the soundness proof of our syntax-directed proof rules to be tractable, and it remains manifestly equivalent to the reference axiomatic model.

Executions in our opax semantics are with respect to an ambient complete candidate execution graph that satisfies the axiomatic model validity predicate (but unconstrained by the thread-local ISA semantics), which is picked non-deterministically at the start. The semantics executes threads individually: there is no substantive interleaving, nor interaction directly between threads, only between single threads and the ambient memory graph. The instructions of each thread execute in order, keeping only thread-local state – the next memory event identifier, the contents of registers, sources of control dependencies, etc. Each instruction acts as an assertion about the existence of a corresponding memory event at a particular position in the ambient execution graph, and the thread is stuck if an appropriate memory event does not exist in the graph (in which case that specific execution is stuck in the opax semantics, but of course the assembly program itself does not get stuck). This explicitly manipulates a non-thread-local graph; but in the logic, we manage to hide this non-thread-locality in normal cases (as we show in §5).

Candidate graphs in which one or more thread(s) get stuck are simply ignored. This is unusual: getting stuck is not an error; it indicates rather that this particular graph is not consistent with the thread-local semantics of instructions. This was inspired by a related approach taken for the Islaris logic [Sammler et al. 2022] for reasoning about sequential Arm-A machine-code, which faced a similar challenge in that rather different context[1]. In a sense, this opax model is merely permuting the order of the usual construction of the axiomatic model: it starts by guessing a valid execution graph (that is, an execution graph that follows the constraints), and then checks that each thread's contribution in the graph does indeed correspond to an execution of the thread.

One could instead try to work directly over Promising-Arm [Pulte et al. 2019], which is also operational enough for our current purposes in the above senses. In Promising-Arm, apart from promises of future writes (which can all be done at the start of execution, and which also inspire our up-front nondeterministic choice of graph) each thread executes in program order; threads interact through a linear history of writes, keeping track of certain integer *timestamps* (indices into the history of writes), which constrain how instructions can interact with the history. Timestamps keep track of lower bounds on the sources of register values (and some whole-thread bounds for barriers), abstracting the set of source events for each. These integer timestamps might be technically easier to work with than graphs, but we found the explicit nodes with explicit edges of the axiomatic model helpful in developing our model of assertions. In a sense, our opax semantics is a reformulation of an axiomatic model made to look more like a promising model, but with the advantage that changes to the axiomatic model apply directly.

Note that, while our opax technically qualifies as an operational semantics, it falls short of most usual expectations of such. In particular, there is no reasonable sense in which it is executable.

By putting an axiomatic model in the required shape, we need a non-standard definition of weakest precondition (§6.1) and a non-standard proof of adequacy (§6.2) (even more so as our threads are executed independently), but we still benefit from more fundamental Iris features like higher-order ghost state, which one would not want to reconstruct.

[1] To reason above the full Arm-A ISA semantics without being overwhelmed with irrelevant detail, Islaris simplified the semantics of each instruction with respect to chosen assumptions, e.g. about Arm-A system register values and alignment facts, using Isla SMT-assisted symbolic execution [Armstrong et al. 2021]. The resulting symbolic traces contain asserts on some paths, which (when they fail to hold) discard those paths from the instruction semantics – which the Islaris instantiation of Iris exploits.

Developing a separation logic directly over an axiomatic memory model has previously been done either using a non-standard semantics of assertions (e.g. RSL, FSL, and GPS, which, as noted by Kaiser et al. [Kaiser et al. 2017, §1.2], requires significant effort), or by defining an equivalent, operational model (e.g. iGPS [Kaiser et al. 2017] and ORC11 [Dang et al. 2020]), which is challenging when the model allows very relaxed behaviour.

3.3 The Third Problem: Structuring the Adequacy Proof

Finally, given a proof in AxSL using our new assertions, we then need an adequacy theorem (§6.2), which, given a family of thread-local proofs in our logic, gives a statement about a whole program in the meta-logic, sound w.r.t. the Arm-A semantics. To prove such an adequacy theorem, we need to address a tension between our proof, which is syntax-directed, and therefore in program order, and synchronisation, which is along rf — even though there can be cycles in $\mathsf{po} \cup \mathsf{rf}$, which prevents doing an induction on it. **Our third key idea** is that, to solve this tension, we can split the proof of adequacy in two phases: first along po, and then along ob. The first phase, along po, uses thread-local resources to establish, for each thread, and for each memory event of that thread, that the flow implication for that event holds. The second phase, along ob, stitches the flow implications together. For example, this second phase walks through the thread of Fig. 6 twice, for the two disjoint components of ob: once from a to c, and once from b to d (with both orderings being possible).

4 THE LANGUAGE

As the focus of this paper is on real-world concurrency rather than realistic instruction set architectures, we consider a simplified language in which to write simple Arm-A concurrency-model programs. However, we give its semantics by elaborating it into the outcome interface type of Sail [Gray et al. 2015][Pulte et al. 2018, §6.1][Pulte 2018, §2.3] (§4.1), translating instructions into sequences of their semantic "microinstructions": primitive register and memory accesses, and Arm-A fences. These are what our logic actually reasons about. Using our basic rules for the interface events, we then give high-level rules for our toy instructions. This means the logic should extend naturally to the full Sail semantics for a large fragment of the Arm-A instruction-set architecture (ISA), using either the Sail-generated Coq definitions for the ISA, or (as in Islaris [Sammler et al. 2022]) the output of the Isla symbolic evaluator for Sail [Armstrong et al. 2021], both of which express the intricate real semantics of instructions in terms of that same outcome interface type.

Our simplified assembly language is shown in Fig. 9. Loads and stores are parameterised by an *ordering strength*, *os*, either plain, release/acquire, or weak-acquire, and a *variety*, *vr*: non-exclusive or exclusive. The output register of a store is used only for the success/fail value of a store exclusive; a dummy register is used for other stores.

4.1 The Elaboration Semantics of Instructions into the Sail Outcome Interface

The Sail outcome interface defines the intra-instruction semantics of each instruction, in isolation from the behaviour of registers and memory. It does this in terms of abstract microinstructions, formally a free monad of effects on *outcomes*, with constructors RegRead, RegWrite, MemRead, MemWrite, etc. Each of these takes the appropriate arguments, and the free monad constructor Next pairs it with a continuation which takes any register or memory read result and gives the subsequent intra-instruction semantics. MemRead $os\ vr\ x\ D$ has type (roughly) Outcome Word, wrapped in the instruction monad IMon A which has constructors $\mathrm{Next}_T : \mathrm{Outcome}\ T \rightarrow (T \rightarrow \mathrm{IMon}\ A) \rightarrow \mathrm{IMon}\ A$ and $\mathrm{Ret} : A \rightarrow \mathrm{IMon}\ A$.

Rather than give an exhaustive definition, we sketch how a few special cases of our instructions elaborate into (a meta-level Coq program over) this Sail outcome interface in Fig. 10. A plain,

$$\begin{array}{lll}
i ::= \mathsf{nop} & \text{instructions} & r \in \mathsf{Reg} \triangleq \{r_0, r_1, \ldots\} \\
\quad \mid\ r := t & \quad \text{register assignment} & v, x \in \mathsf{Word} \triangleq 0..2^{64}-1 \\
\quad \mid\ \mathsf{br}\ x & \quad \text{branch to address } x & op \quad ::= \quad + \mid - \mid \times \\
\quad \mid\ \mathsf{bne}\ t\ x & \quad \text{conditional branch} & \quad t \quad ::= \quad v \mid r \mid t_1\ op\ t_2 \\
\quad \mid\ r := \mathsf{ldr}_{os,vr}\ [t_{\text{addr}}] & \quad \text{memory load} & os \quad ::= \quad \mathsf{plain} \mid \mathsf{relacq} \mid \mathsf{weakacq} \\
\quad \mid\ r := \mathsf{str}_{os,vr}\ [t_{\text{addr}}]\ t_{\text{data}} & \quad \text{memory store} & vr \quad ::= \quad \mathsf{nexcl} \mid \mathsf{excl} \\
\quad \mid\ \mathsf{dmb\ sy} \mid \mathsf{dmb\ st} \mid \mathsf{dmb\ ld} \mid \mathsf{isb} & \quad \text{Arm-A fences} &
\end{array}$$

$$\begin{array}{lcl}
r := \mathsf{ldr}\ [t_{\text{addr}}] & \triangleq & r := \mathsf{ldr}_{\mathsf{plain,nexcl}}\ [t_{\text{addr}}] \\
r := \mathsf{ldar}\ [t_{\text{addr}}] & \triangleq & r := \mathsf{ldr}_{\mathsf{relacq,nexcl}}\ [t_{\text{addr}}] \\
\mathsf{stlr}\ [t_{\text{addr}}]\ t_{\text{data}} & \triangleq & r := \mathsf{str}_{\mathsf{relacq,nexcl}}\ [t_{\text{addr}}]\ t_{\text{data}}
\end{array}$$

Fig. 9. Instructions and syntactic sugar

$$\begin{array}{lll}
[\![r := \mathsf{ldr}\ [t_{\text{addr}}]]\!] & \triangleq & \mathbb{T}[\![t_{\text{addr}}]\!] \gg= \lambda x.\ \mathsf{Next}\ (\mathsf{MemRead\ plain\ nexcl}\ x\ \emptyset) \\
 & & (\lambda v.\ \mathsf{Next}\ (\mathsf{RegWrite}\ r\ v\ \langle \emptyset, \{0\} \rangle)\ (\lambda().\ \mathsf{Ret}\ ()))\ \gg \mathsf{IncPC} \\
[\![\mathsf{stlr}\ [t_{\text{addr}}]\ t_{\text{data}}]\!] & \triangleq & \mathbb{T}[\![t_{\text{addr}}]\!] \gg= \lambda x.\ \mathbb{T}[\![t_{\text{data}}]\!] \gg= \lambda v. \\
 & & \mathsf{Next}\ (\mathsf{MemWrite\ rel\ nexcl}\ x\ v\ (\mathbb{D}[\![t_{\text{addr}}]\!])\ (\mathbb{D}[\![t_{\text{data}}]\!]))\ (\lambda().\ \mathsf{Ret}\ ())\ \gg \mathsf{IncPC} \\
[\![\mathsf{bne}\ t\ x]\!] & \triangleq & \mathbb{T}[\![t]\!] \gg= \lambda v.\ \mathsf{Next}\ (\mathsf{BranchAnnounce}\ x\ \mathbb{D}[\![t]\!]) \\
 & & \left(\lambda().\ \begin{array}{l} \mathsf{if}\ v = 0\ \mathsf{then\ IncPC} \\ \mathsf{else\ Next}\ (\mathsf{RegWrite\ pc}\ x\ \langle \emptyset, \emptyset \rangle)\ (\lambda().\ \mathsf{Ret}\ ()) \end{array} \right) \\
\mathsf{IncPC} & \triangleq & \mathsf{Next}\ (\mathsf{RegRead\ pc})\ (\lambda v.\ \mathsf{Next}\ (\mathsf{RegWrite\ pc}\ (v+4)\ \langle \emptyset, \emptyset \rangle)\ (\lambda().\ ()))
\end{array}$$

$$\begin{array}{lll@{\qquad}lll}
\mathbb{T}[\![v]\!] & \triangleq & \mathsf{Ret}\ v & \mathbb{D}[\![v]\!] & \triangleq & \emptyset \\
\mathbb{T}[\![r]\!] & \triangleq & \mathsf{Next}\ (\mathsf{RegRead}\ r)\ \mathsf{Ret} & \mathbb{D}[\![r]\!] & \triangleq & \{r\} \\
\mathbb{T}[\![t_1\ op\ t_2]\!] & \triangleq & \mathbb{T}[\![t_1]\!] \gg= \lambda v_1.\ \mathbb{T}[\![t_2]\!] \gg= \lambda v_2.\ \mathsf{Ret}\ (v_1\ \mathbb{O}[\![op]\!]\ v_2) & \mathbb{D}[\![t_1\ op\ t_2]\!] & \triangleq & \mathbb{D}[\![t_1]\!] \cup \mathbb{D}[\![t_2]\!]
\end{array}$$

Fig. 10. A few cases of the elaboration of our simplified assembly into the outcome interface (eliding some details). The computation of intra-instruction dependencies is highlighted.

non-exclusive load $r := \mathsf{ldr}\ [t_{\text{addr}}]$, into register r from address t_{addr}, elaborates into a plain (and non-exclusive) memory read from that address with no dependencies (indicated by $\emptyset$), the value of which is bound to v, followed by a register write to r of v with no register dependency ($\emptyset$ on the left) and a dependency on the 0th MemRead of the instruction ($\{0\}$ on the right), and a program counter increment (which we write with a bind $\gg=$ to be systematic). A non-exclusive store release $\mathsf{stlr}\ [t_{\text{addr}}]\ t_{\text{data}}$, at t_{addr} of t_{data}, elaborates into the elaboration of the evaluation of terms t_{addr} and t_{data} to some x and v, using the auxiliary function $\mathbb{T}[\![-]\!]$, followed by a release (and non-exclusive) memory write to x of v with address data dependencies given by the auxiliary function $\mathbb{D}[\![-]\!]$ (in our simplified language, dependencies can be computed from the syntax) and again a PC increment (the dummy register is not mentioned). We use the Sail interface BranchAnnounce outcome to capture dependencies of branches, which we elaborate into evaluation of their condition, followed by, depending on whether the condition holds (using the conditional of the meta-language), either a write of the given address x to the program counter, or a normal program counter increment.

4.2 The Conventional Axiomatic Concurrency Model Semantics

A program working over the Sail outcome interface can then be glued onto a memory model, either operational, axiomatic, or promising. For an axiomatic memory model, this is usually done by recursively computing the set of thread-local instruction-semantics pre-executions of (the control-flow unfoldings of) each thread, allowing arbitrary concrete values for register and memory reads, and then taking the cartesian product of these sets. For each such pre-execution, one enumerates the set of candidate executions, decorating the pre-execution with rf and co relations (unconstrained except for some well-formedness properties). Finally, one filters those with the axiomatic-model validity predicate.

4.3 Our Opax Concurrency Model Semantics

However, as discussed in §3.2, it is not clear how to define a syntax-directed program logic over this style of semantics. Hence we reformulate the combination of axiomatic model and instruction semantics in our novel *opax* semantics, mixing operational and axiomatic styles.

The opax semantics works in two phases. In the first phase, execution of a whole program starts by guessing a complete candidate execution graph X for the program. This execution graph has to be well-formed and satisfy the axiomatic-model validity predicate, but is otherwise unconstrained, and in particular is for now unrelated to the program itself. In the second phase, each thread is executed independently, with interaction only via the execution graph.

For simplicity we assume a fixed instruction memory I, which is simply a map from addresses to opcode values, and n threads, with initial program counter values $c_1, \ldots, c_n$. Each thread state s is either Ctd T ("continued"), which represents an ongoing thread execution, or Done R, which represents a completed thread execution. Here T is a pair $\langle p, R\rangle$ and R is a tuple $\langle regs, e_{\mathrm{po}}, srcs_{\mathsf{ctrl}}, e\rangle$, together comprising: the remaining microinstruction program p in the Sail outcome interface for the current instruction, a register state $regs$, the identifier of the previous event e_{po} (or initially None), the set $srcs_{\mathsf{ctrl}}$ of sources of control dependencies, and the next identifier e. For each thread tid, $s_{\mathrm{init}}(c_{tid})$ is its initial thread state, with $regs(\mathrm{pc}) = \langle c_{tid}, \emptyset\rangle$ and microinstruction program Ret () (before the first instruction has started). Execution of a thread terminates when it has finished execution of the current microinstruction program and the program counter points outside of instruction memory. Thread transitions $s \xrightarrow{tid,X,I}_{\mathsf{h}} s'$ are indexed by the thread ID, execution graph, and instruction memory.

Successful whole system execution requires each thread to execute to completion independently:

$$\textsc{Whole-system-execution}\quad \frac{(s_{\mathrm{init}}\ c_1) \xrightarrow{1,X,I}{}^{*}_{\mathsf{h}} \mathsf{Done}\ \langle\_\rangle \quad \ldots \quad (s_{\mathrm{init}}\ c_n) \xrightarrow{n,X,I}{}^{*}_{\mathsf{h}} \mathsf{Done}\ \langle\_\rangle}{\langle c_1 \parallel \ldots \parallel c_n, I,\ X\rangle \rightarrow_{\mathrm{tp}} \checkmark}$$

A stuck thread is not an error state, but rather indicates that the guessed execution graph does not correspond to this program. Rule Whole-system-execution of our semantics and our definition of weakest precondition (see §6.1) ignore them. We sketch selected rules of the thread operational semantics in Fig. 11. A thread executes by executing the current microinstruction program until it ends, and then (rule H-reload) fetching the next instruction at the address in register pc by looking it up in I, and decoding it into a new microinstruction program, until pc is outside the instruction memory (rule H-term), at which point there must not be further events by this thread in the graph. A MemRead microinstruction can execute (rule H-mem-read) only when there is a corresponding memory read event in the execution graph; otherwise, this instruction (and thus this thread) is stuck. This event has to be of the appropriate type, and has to have the appropriate po, addr and ctrl edges to it, as induced by the thread state (by the po predecessor e_{po}, the set of data dependencies d_{addr}, and the control dependency sources $srcs_{\mathsf{ctrl}}$, respectively). A RegWrite microinstruction can similarly execute (rule H-reg-write) only when there is a corresponding register write event in the execution graph. The graph register write event needs to agree with the thread-local register state, both on value and on dependencies, as well as on edges. (This register event is not used in the current axiomatic memory model, which instead uses primitive dependency relations, but the interface includes it to support operational models and other axiomatic models.)

An event identifier comprises a thread identifier (zero being reserved for the 'initial' thread that contains all the initial writes), an instruction counter (incremented with next-i), and an intra-instruction event counter (incremented with next-e).

$$\textsc{H-mem-read}\quad \frac{\begin{gathered}X.\mathsf{lab}(e) = \mathsf{R}_{os,vr}\ x\ v \qquad \langle e_{\mathsf{po}}, e\rangle \in to(e, X.\mathsf{po}_1) \qquad \{\langle e_d, e\rangle \mid e_d \in srcs_{\mathsf{ctrl}}\} = to(e, X.\mathsf{ctrl})\\ \{\langle e_d, e\rangle \mid r \in d_{\mathsf{addr}}.\mathsf{regs} \wedge regs(r) = \langle \_, srcs_d\rangle \wedge e_d \in srcs_d\} = to(e, X.\mathsf{addr})\end{gathered}}{\mathsf{Ctd}\ \langle \mathsf{Next}\ (\mathsf{MemRead}\ os\ vr\ x\ d_{\mathsf{addr}})\ K, \langle regs, e_{\mathsf{po}}, srcs_{\mathsf{ctrl}}, e\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle K\ v\ \{e\}, \langle regs, e, srcs_{\mathsf{ctrl}}, \mathsf{next\text{-}e}(e)\rangle\rangle}$$

$$\textsc{H-reg-write}\quad \frac{X.\mathsf{lab}(e) = \mathsf{RegWrite}\ r\ v \qquad \langle e_{\mathsf{po}}, e\rangle \in to(e, X.\mathsf{po}_1) \qquad srcs'_{\mathsf{reg}} = \bigcup_{\{srcs_d \ \mid\ r \in d_{\mathsf{data}}.\mathsf{regs} \wedge regs(r) = \langle \_, srcs_d\rangle\}} srcs_d \cup d_{\mathsf{data}}.\mathsf{m}}{\mathsf{Ctd}\ \langle \mathsf{Next}\ (\mathsf{RegWrite}\ r\ v\ d_{\mathsf{data}})\ K, \langle regs, e_{\mathsf{po}}, ?R, e\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle K\ (), \langle regs[r \mapsto \langle v, srcs'_{\mathsf{reg}}\rangle], e, ?R, \mathsf{next\text{-}e}(e)\rangle\rangle}$$

$$\textsc{H-reload}\quad \frac{\begin{gathered}regs(\mathsf{pc}) = \langle x, \_\rangle \qquad I(x) = opcode\\ \mathsf{decode}(opcode) = p\end{gathered}}{\mathsf{Ctd}\ \langle \mathsf{Ret}\ (), \langle regs, ?R, e\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle p, \langle regs, ?R, \mathsf{next\text{-}i}(e)\rangle\rangle} \qquad \textsc{H-term}\quad \frac{\begin{gathered}regs(\mathsf{pc}) = \langle x, \_\rangle \qquad x \notin \mathsf{dom}(I)\\ \nexists e'.\ \langle e, e'\rangle \in X.\mathsf{po}_1\end{gathered}}{\mathsf{Ctd}\ \langle \mathsf{Ret}\ (), \langle regs, ?R\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Done}\ \langle regs, ?R\rangle}$$

Fig. 11. Selected reduction rules of our operationalised semantics. We write $?R$ to stand for the rest of a R. We write $to(e, \mathcal{R})$ for the set of edges of type $\mathcal{R}$ with source e, and $\mathcal{R}_1$ for the transitive reduction of $\mathcal{R}$ (that is, $\mathcal{R}$ without transitively induced edges).

The guessing step and the fact that executions can get stuck mean that this model is not executable as such, but this is not problematic for a logic. First, we treat discarded executions by ignoring stuck states in the definition of weakest preconditions. Second, the guessing does not appear in the definition of weakest preconditions, which takes the guessed graph as a parameter; instead, the guessing is handled by a quantification in the adequacy theorem, when the proofs of the individual threads are combined.

Infinite executions. Handling infinite executions in memory models exhibiting load buffering is currently an open problem. The problem manifests in axiomatic models in the form of an infinite regress, where an event is justified by a program-order-later event, itself justified by another program-order-later event, ad infinitum, without an eventual grounding, but because this is not a cycle, most axiomatic models do not reject this kind of execution. The same underlying problem appears in the promising and operational models of Arm-A under a different guise. We do not attempt to tackle this problem, and our opax semantics sticks to the axiomatic model as-is.

5 THE LOGIC

The AxSL proof rules (and the Hoare triples used in them) are defined at two abstraction levels: the underlying rules are for microinstructions, and are proven sound against the semantics of Hoare triples described in §6.1, while the high-level rules explicitly used in proofs of examples are for the surface instructions of Fig. 9, derived from the former by reasoning about instruction elaboration (Fig. 10). We first explain the AxSL notion of *protocols*, used in both to enable thread-local rely/guarantee reasoning (§5.1), and explain one of the key microinstruction rules, for MemRead (§5.2). We then explain how AxSL can be used to reason compositionally, using two key examples: versions of load buffering (§5.3) and message-passing (§5.4), describing our high-level proof rules along the way. We highlight why the proofs go through, and how they fail (as desired) if the necessary synchronisation is removed. These examples also demonstrate two styles of proof: a high-level proof for LB that abstracts physical state with simple, easy-to-use ghost state, demonstrating the convenience of our logic; and a low-level proof for MP working explicitly with graph facts (reflected as ghost state), showing how one can tackle subtle reasoning, that depends on intricacies of the memory model, if needed. Finally, we describe how the logic we have seen so far has been extended, with only minor changes, with proof rules for Arm-A exclusives (§ 5.5), and briefly describe other examples we verified with AxSL (§5.6).

5.1 Protocols

An AxSL protocol Φ is a rely/guarantee-style protocol that enables thread-local reasoning by expressing the intended resource transfer across an entire program. Our Hoare triples are parameterised by a protocol, which should be agreed upon by all threads, like invariants in CSL. Our triples ensure the protocol holds at every event, and enable resource transfer between events. In particular, as we see in §5.2, every read event obtains the resource specified by Φ from the external write that it is reading from, and every external write must supply that resource. An AxSL protocol takes as arguments an address x, a value v, and an event ID e. The event ID e is the write to x of v written to for stores, and read from for loads. This event ID argument e is used for explicit reasoning about the execution graph, as illustrated in §5.4. For example, it makes it possible to state “there exists a write to a certain address of a certain value that is lob-before e” using our library of graph ghost state properties. For simpler cases, this last argument can be elided, as we have so far. In addition, as we also have so far, we can define per-location protocols, and combine them pointwise.

5.2 Our Logic for Microinstructions

We now explain one of the key proof rules used for reasoning about the language of microinstructions: rule HT-MICRO-MEMREAD-RDEP-EXT in Fig. 12, for a MemRead with ordering strength os, variety vr, address x, and address dependencies d. To keep the exposition manageable, the rule is specialised to a read from a distinct thread (an external read), with empty intra-instruction dependencies (i.e. only syntactic register dependencies). We illustrate this rule schematically in Fig. 13. The rule is proved sound against our opax semantics, specifically the H-MEM-READ rule. We present the rule for a MemRead microinstruction in isolation, leaving the plumbing into its continuation implicit.

HT-MICRO-MEMREAD-RDEP-EXT

$$\left\{\begin{array}{l}
(1)\mathsf{NoLocalWrites}(x) * (2)d = (\mathrm{dom}(regs), \emptyset) * \\
(3)\mathsf{PoPred}(e_{\mathrm{po}}) * (4)\mathsf{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\
(5) \mathop{\Large\ast}\limits_{(r \mapsto v@E) \in regs} r \overset{\mathrm{r}}{\mapsto} v@E * (6) \mathop{\Large\ast}\limits_{(e_{\mathrm{lob}} \mapsto P_{\mathrm{lob}}) \in m} (e \hookrightarrow P_{\mathrm{lob}}) * \\
\forall e, v, e_{w}. \left(\begin{array}{l}
(7)\mathsf{GraphFacts}(e, os, vr, x, v, e_{w}, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) \mathrel{-\!\!*} \\
((8)\mathsf{Lob}(\mathrm{dom}(m), e) * (9)\mathsf{Flow}_{\Phi}(e, x, v, e_{w}, m, P))
\end{array}\right)
\end{array}\right\}$$

$$\mathsf{MemRead}\ os\ vr\ x\ d$$

$$\left\{ v, E. \begin{array}{l}
\exists e, e_{w}. E = \{e\} * (10)\mathsf{NoLocalWrites}(x) * (11)\mathsf{PoPred}(e) * (12)\mathsf{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\
(13) \mathop{\Large\ast}\limits_{(r \mapsto v@E) \in regs} r \overset{\mathrm{r}}{\mapsto} v@E * (14)\mathsf{GraphFacts}(e, os, vr, x, v, e_{w}, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) * \\
(15) e \hookrightarrow P(x, v, e_{w})
\end{array}\right\}_{tid,\Phi}$$

Fig. 12. A proof rule of AxSL for the MemRead microinstruction, specialised for a thread that has no writes to the location read. The user provides m, a thread-local map from events to the resources consumed.

Our Hoare triple for MemRead has the form $\{P\}\mathsf{MemRead}\ \mathit{os}\ \mathit{vr}\ x\ d\ \{v, E.\ Q\}_{tid,\Phi}$, which states that, for a MemRead on thread *tid* following protocol Φ, if one provides the resources specified in the precondition P, then the MemRead results in the updated resources Q of the postcondition, which can refer to the resulting read value v and dependency set E (the set of event IDs that v stems from). In this rule $E = \{e\}$, where e is the existentially quantified event ID of the associated memory read event. These v and E are then passed to the continuation, as in the opax rule, to continue reasoning about the thread.

The rule has two main aspects, as do the other low-level rules for MemRead and MemWrite: low-level graph reasoning, and high-level resource transfer.

First, for directly conducting graph reasoning with respect to the axioms of the memory model, the postcondition gives us ghost state describing what we learn about the execution graph as a result of executing the MemRead (as per the premises of opax rule H-MEM-READ), including the existence of a corresponding memory read event e and its incoming edges, which we collect as GraphFacts. The GraphFacts are mentioned twice in the rule: they appear, as expected, existentially quantified as facts that we learn in the postcondition as (13), but they also appear in the precondition (7), in hypothetical reasoning, as an assumption that may be used to establish that there will be lob edges to the new event sufficient to satisfy the flow implication. (This style of specification, using a separation implication $-\!\!*$ in the antecedent, is similar to how the weakest-precondition rules are often formulated in Iris, see, e.g., the rule for store in [Jung et al. 2018, Page 45].)

Second, to support high-level reasoning via resource transfer, when some of the incoming edges are in ob, this rule allows us to reason about resources flowing to e along those edges. If there is such an incoming edge, say from e' to e, and we have an $e' \hookrightarrow P$, then the flow implication can use P in its premise. In total, the resources that flow into e, and thus are considered in the flow implication for e, consist of (the separating conjunction of) all such resources that flow along lob edges, as collected in the partial event-to-resource map m (for thread-internal resource flow), combined with the resources flowing from external events (here, the quantified external write e_w that e is reading from), as specified by the protocol Φ.

The first two clauses of the precondition capture the specialisation mentioned above: (1) $\mathsf{NoLocalWrites}(x)$ captures the fact that there are no thread-local writes on the same address x up until this point in the program order; with this in hand, one knows that only external writes can be read from by this MemRead, and thus resource transfer along obs is possible. This fact is unchanged after the MemRead, and is thus restored by the postcondition as (10). (2) $d = (\mathrm{dom}(\mathit{regs}), \emptyset)$ requires that this MemRead depends on *exactly* the registers in the domain of the (partial) register file *regs* of (5) (non-involved registers can be framed off to apply this rule), and not on any intra-instruction memory read (the $\emptyset$ of event IDs). The collection of register points-to for *regs* of (5) is unchanged and thus restored in the postcondition as (13) (although, in the elaboration of an instruction, a MemRead is typically followed by a RegWrite).

Bookkeeping assertions (2) $\mathsf{PoPred}(e_{\mathsf{po}})$ and (3) $\mathsf{CtrlPreds}(\mathit{srcs}_{ctrl})$ capture the e_{po} and $\mathit{srcs}_{\mathsf{ctrl}}$ parts of the opax thread state T. Intuitively, they keep track of the events that will become the po and ctrl predecessors of the memory event e that will be induced to the next microinstruction, and thus allow us to conclude facts about new incoming edges, e.g. e_{po} po e, that are included in GraphFacts. They get updated accordingly in the postcondition, as (11) and (12).

Assertion (6) $*_{(e_{\mathsf{lob}} \mapsto P_{\mathsf{lob}}) \in m}(e \hookrightarrow P_{\mathsf{lob}})$ collects the thread-local resources in m for the premise of the flow implications. m is user-supplied, and is constrained by assertion (8) in the hypothetical reasoning, which requires that an event e can only occur in the domain of m when there will be (given the graph facts (7)) an lob edge to the new MemRead event.

Finally, as a result of the hypothetical reasoning on the last line of the precondition, we get the (user-supplied) result $P(x, v, e_w)$ of the flow implications (9), tied to the new memory event e, as

(15), where the flow implication is defined as:

$$\mathsf{Flow}_{\Phi}(e, x, v, e_w, m, P) \triangleq \left(\left(\mathop{\ast}_{(\_\mapsto P_{\mathsf{lob}})\in m} P_{\mathsf{lob}}\right) \ast \Box(\Phi(x, v, e_w))\right) \Rrightarrow P(x, v, e_w)$$

This flow implication is an instance of the general definition of the flow implication, specialised for a read, which receives a resource. Symmetrically, in a rule for MemWrite, the Φ is on the right side of the view shift, so that this resource can be sent away.

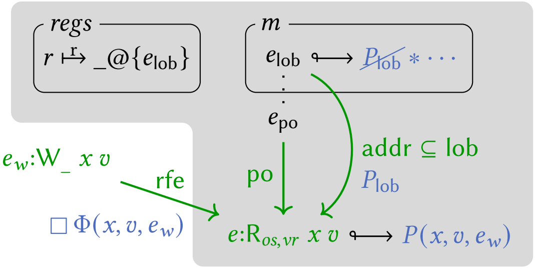
regs
$r \overset{\mathrm{r}}{\mapsto} \_@\{e_{\mathsf{lob}}\}$
m
$e_{\mathsf{lob}} \hookrightarrow \cancel{P_{\mathsf{lob}}} \ast \cdots$
e_{po}
po
addr ⊆ lob
P_{lob}
e_w:W_ x v
rfe
$\Box\,\Phi(x, v, e_w)$
e:R$_{os,vr}$ x $v \hookrightarrow P(x, v, e_w)$

Fig. 13. A visualisation of HT-MICRO-MEMREAD-RDEP-EXT for external read event e, with logical annotations and newly gained graph facts highlighted. The grey area depicts the resources and the thread we are reasoning about locally. The protocol specifies the persistent resources $\Box\,\Phi(x, v, e_w)$ passed along the rfe edge to e. Local events e_{lob}, which are shown to become lob predecessors due to the use of some register r, let the corresponding P_{lob} resource, previously tied to e_{lob}, flow along the newly-learned lob edge, to be combined with the protocol, and the result $P(x, v, e_w)$ gets tied to e.

5.3 Our Surface Logic, on an Example

Moving from microinstructions to the instructions composed out of them, we can write a derived high-level proof rule for each instruction, and these high-level rules can be further specialised to specific programming idioms and their assumptions. The Arm-A memory model is intrinsically complex, so there is no 'perfect' rule that is both simple and general, but we can derive specialised yet reasonably general high-level rules that make reasoning practical. We demonstrate this on LB+artificialdata+data (Fig. 14), a version of the LB litmus test with an artificial (but still architecturally respected) data dependency on Thread 1, and a normal data dependency on Thread 2. This version of LB is interesting because this specification cannot be proved just using an invariant about the *values* of the write, as it can for LB+datas [Jeffrey and Riely 2016, §6]: it requires reasoning about the *order* of the writes.

Specification. We would like to show that the example exhibits no load buffering behaviour on Arm-A, i.e., that Thread 1 cannot read 1 (while Thread 2 can read either the initial value 0, or the value 1 that Thread 1 writes), as in the Fig. 14 specification. We give two versions of the postcondition: one still involving tied assertions, corresponding to the final state of the threads, and one when the assertions have been pulled out, as used in our formal definition of weakest preconditions, as we describe in §6.1.

Thread 1	Thread 2
$\{r_1 \overset{\mathrm{r}}{\mapsto} \_ \ast \text{some ghost state}\}$	$\{r_2 \overset{\mathrm{r}}{\mapsto} \_ \ast \text{some ghost state}\}$
a: $r_1 :=$ ldr $[x]$	c: $r_2 :=$ ldr $[y]$
b: str $[y]$ $(1 + r_1 - r_1)$	d: str $[x]$ r_2
$\{\exists v_1.\, r_1 \overset{\mathrm{r}}{\mapsto} v_1@\{a\} \ast a \looparrowright (v_1 = 0)\}$	$\{\exists v_2.\, r_2 \overset{\mathrm{r}}{\mapsto} v_2@\{c\} \ast c \looparrowright (v_2 = 0 \lor v_2 = 1)\}$
$\{r_1 \overset{\mathrm{r}}{\mapsto} 0@\_\}$	$\{\exists v_2.\, r_2 \overset{\mathrm{r}}{\mapsto} v_2@\_ \ast (v_2 = 0 \lor v_2 = 1)\}$

Fig. 14. LB+artificialdata+data and its (informal) specification

Protocol. The first step of the proof is to come up with an appropriate protocol Φ that abstracts the interference of the threads, and thus enables thread-modular reasoning. For this LB shape, it suffices to transfer the information that the write of 1 by Thread 1 has been executed, in the sense that this write is ob-before the event to which this information is tied. To capture this logically, we use a simple form of ghost state: the 'oneshot resource algebra' of Iris [Jung et al. 2018, §2]. The oneshot has two states: pending represents the exclusive permission to make a decision to choose a value, and shot(v) represents the information that the decision has been made to choose value v. In particular, $\boxed{\text{pending}} * \boxed{\text{pending}}$ is a contradiction, and so is $\boxed{\text{pending}} * \boxed{\text{shot}(v)}$, but we can view-shift pending into shot(v), $\boxed{\text{pending}} \Rrightarrow \boxed{\text{shot}(v)}$, to express the logical decision to commit to v. Using this, we can formalise our protocol: for both locations x and y, either the value is 0, or it is 1, in which case we also have $\boxed{\text{shot}(1)}$.

$$\Phi(\_, v, \_) \triangleq v = 0 \vee (v = 1 * \boxed{\text{shot}(1)})$$

Thread 1:

1 $\{r_1 \overset{\mathrm{r}}{\mapsto} \_ * \mathsf{NoLocalWrites}(x) * \boxed{\text{pending}} * \ldots\}$

2 $\left(v_1 = 0 \vee \left(v_1 = 1 * \boxed{\text{shot}(1)}\right)\right) * \boxed{\text{pending}}$

$\Rrightarrow$

3 $v_1 = 0 * \boxed{\text{pending}}$

4 a: r_1 := ldr $[x]$

5 $\left\{\exists v_1.\, r_1 \overset{\mathrm{r}}{\mapsto} v_1 @\{a\} * a \looparrowright \left(v_1 = 0 * \boxed{\text{pending}}\right) * \ldots\right\}$

6 $\boxed{\text{pending}}$

$\Rrightarrow$

7 $\boxed{\text{shot}(1)}$

8 b: str $[y]$ $(1 + r_1 - r_1)$

9 $\{\exists v_1.\, r_1 \overset{\mathrm{r}}{\mapsto} v_1 @\{a\} * a \looparrowright (v_1 = 0)\}$

Thread 2:

10 $\{r_2 \overset{\mathrm{r}}{\mapsto} \_ * \mathsf{NoLocalWrites}(y) * \ldots\}$

11 c: r_2 := ldr $[y]$

12 $\{\exists v_2.\, r_2 \overset{\mathrm{r}}{\mapsto} v_2 @\{c\} * c \looparrowright \Phi(y, v_2, c) * \ldots\}$

13 d: str $[x]$ r_2

14 $\left\{\exists v_2.\, r_2 \overset{\mathrm{r}}{\mapsto} v_2 @\{c\} * c \looparrowright \left(v_2 = 0 \vee \left(v_2 = 1 * \boxed{\text{shot}(1)}\right)\right)\right\}$

Fig. 15. Proof sketch of LB+artificialdata+data.

Proof sketch. Using this protocol, the proof follows the sketch in Fig. 15. On line 1, we give Thread 1 the exclusive permission pending to choose a value, which it will need when it writes 1; moreover, we will use pending in the flow implication of the load from x of line 4 to show that it must read 0[2]. Line 2 states the incoming resources of the flow implication: the disjunction obtained from the protocol, and the pending from the context. In the flow implication, we can then do a case analysis on the disjunction, and in the case of $v_1 \neq 0$, we can derive a contradiction by combining pending with shot(v); therefore, we must be in the case $v_1 = 0$, still with pending in hand, as per line 3. On line 5, because we have used the pending from the context in the flow implication for a, we get it back, but tied to a. This deals with the load. Now, for the store on line 8, we need, as part of the flow implication, to establish the protocol for the written value, 1. For our protocol,

[2] We also start Thread 1 with the knowledge that it has made no writes to location x, and symmetrically Thread 2 to y, to exclude an internal reads-from. Internal reads-from does not imply ob on Arm-A, and thus has a significantly weaker premise for its flow implication, without $\Phi(x, v_1, \_)$.

we have to take the second disjunct, and so we have to provide $\mathsf{shot}(1)$. Because of the artificial dependency, the flow implication gives access to the resources tied to a, and thus to pending, as per line 6. The pending can be view-shifted, as part of the flow implication, into any $\mathsf{shot}(v)$, and thus in particular into $\mathsf{shot}(1)$, as per line 7. This satisfies the flow implication, and thus concludes the proof sketch for Thread 1.

Thread 2 relies on the same dependencies, but is much simpler: given our protocol Φ, we have $\Phi(y, v_2, c) = \Phi(x, v_2, d)$, so Thread 2 merely forwards this $\Phi(\_, v_2, \_)$ from the load to the store, which the flow implication for the write allows because of the data dependency.

Abstraction. Using the oneshot resource algebra allows us to reason thread-modularly: the proof of Thread 1 does not involve any graph reasoning about the intricacies of the Arm-A memory model, merely reasoning about abstract state. This is already useful for this small proof sketch (and the corresponding mechanised proof). Thread 2 does not require any inspection of the value or the resource being forwarded, merely plumbing through the flow implication, and the derivation of the contradiction in the impossible case of the load of Thread 1 relies on simple ghost theory. Moreover, the proof is quite flexible: the proof sketch only requires trivial modifications if (e.g.) we replace the store of $1 + r_1 - r_1$ by a store of r_1.

Soundness. The proof sketch above crucially relies on the artificial data dependency of the store on the load, as it should: without it, if the store were merely $\mathsf{str}\ [y]\ 1$, it could be executed out of order with respect to the load, thus making the relaxed load behaviour observable. More concretely, without the dependency, the flow implication for the store would not include the resource tied to the memory read event a, and would therefore be of the shape $\top \Rrightarrow \boxed{\mathsf{shot}(1)}$, which is not provable.

Instructions in memory. Before looking at the proof rules that make the proof sketch above precise, we make our treatment of instructions precise. As usual, reasoning about a machine with instructions in (and fetched from) specific addresses in memory, rather than a language with an abstract syntax of statements, causes a slight impedance mismatch with Hoare logic: the thread state does not include instructions, but merely the address of the "current" instruction. However, a normal-looking Hoare triple can be recovered by using some indirection [Myreen et al. 2007; Myreen and Gordon 2007; Myreen et al. 2008][Myreen 2009, §3.4][Erbsen et al. 2021, §4.3][Liu et al. 2023]. We use Hoare triples for presentation, but use weakest preconditions in our formalisation. Our Hoare triples for a single instruction i are of the form $\{P\}\ a{:}\ i\ \{Q\}_{tid,I,\Phi}$, where a is the address of the instruction. In our rules, P will include $\mathsf{pc} \overset{\mathsf{r}}{\mapsto} a@\_$, and Q include $\mathsf{pc} \overset{\mathsf{r}}{\mapsto} a'@\_$ for the appropriate a' — which is $a + 4$ except for branch instructions. For presentation purposes, for programs without branching (and thus no looping), we can conflate instruction instances with instructions (as we have so far), and thus conflate instruction identifiers a, b, c, etc. with numerical addresses for instructions in memory $a, a + 4, a + 8$, etc. For a language where an instruction instance leads to a single memory event (as we have so far), we can conflate instruction instance identifiers with memory event identifiers. In other cases, we use the counters of the operational semantics, although they can often be quantified over in reasoning rather than considered in detail, merely keeping the information that they are smaller than the current counter (and thus po-before the current event).

Proof rules.

Load. The proof rule that we use for the load in both threads, HT-INS-LDR-PLN-EXT (Fig. 16), is specialised to the instruction: a plain, non-exclusive load with an immediate address x. It is further specialised to the assumption that there are no prior writes to x by this thread, as otherwise the memory model guarantees no synchronisation and thus makes resource transfer unsound.

The first line of the precondition deals with bookkeeping of the po-predecessor and the program counter. The second line requires NoLocalWrites to x. The third line requires ownership of the register r that will be written to by the load, and requires the flow implication for this instruction, which flows the protocol Φ for this address to the R that will be tied to this event — with the appropriate address, value, and memory event parameters. The postcondition then updates the bookkeeping of line 1 accordingly, and keeps the fact that this thread has no writes to x. The key part of the postcondition, highlighted on the last line, is that R is then tied the new memory read event e, which is the source of the contents of register r.

Store. The proof rule that we use for the store in Thread 1, HT-INS-STR-PLN-ARTIFICIALDATA, is similarly specialised to the instruction: a plain, non-exclusive store with an immediate address x and an artificial value dependency on register r with result v. It is further specialised to the assumption that there is an assertion P that is tied to the source of register r.

Again, the first lines of the pre- and postcondition deal with bookkeeping. The second lines deal with the latest write to the address. The key of the precondition, highlighted on line 3, requires (1) ownership of register r together with the knowledge that its source is some memory event e_d, (2) P is tied to the source memory event e_d, and (3) the flow implication, which flows the resource P into the protocol for the written value. The postcondition is then just bookkeeping, P having been consumed by the instruction.

The proof obligation for the protocol merely requires $\Box\,\Phi(\ldots)$, as protocols only transfer persistent (duplicable) resources. Transfer of non-duplicable resources can be implemented on top with some indirection using the expressive ghost state of Iris, as sketched in §5.5.

The proof rule for the store in Thread 2 is almost identical, merely requiring knowing the value v' of register r, and the flow implication requiring the protocol be established for that value, $\Phi(x, v', e)$.

HT-INS-LDR-PLN-EXT

$$\left\{\begin{array}{l} \mathsf{PoPred}(e_{\mathrm{po}}) * \mathsf{pc} \overset{\mathrm{r}}{\mapsto} a@\_ * \\ \mathsf{NoLocalWrites}(x) * \\ r \overset{\mathrm{r}}{\mapsto} \_ * \forall e\ v\ e_w.\, (\Box\,\Phi(x, v, e_w) \Rrightarrow R(x, v, e_w)) \end{array}\right\}$$

$$a\colon r := \mathsf{ldr}\ [x]$$

$$\left\{\exists e, v, e_w.\ \begin{array}{l} \mathsf{PoPred}(e) * \mathsf{pc} \overset{\mathrm{r}}{\mapsto} (a+4)@\_ * \\ \mathsf{NoLocalWrites}(x) * \\ e \hookrightarrow (R(x, v, e_w)) * r \overset{\mathrm{r}}{\mapsto} v@\{e\} \end{array}\right\}_{tid,\Phi}$$

HT-INS-STR-PLN-ARTIFICIALDATA

$$\left\{\begin{array}{l} \mathsf{PoPred}(e_{\mathrm{po}}) * \mathsf{pc} \overset{\mathrm{r}}{\mapsto} a@\_ * \\ (\mathsf{NoLocalWrites}(x) \vee \mathsf{LastLocalWrite}(x, \_)) * \\ r \overset{\mathrm{r}}{\mapsto} \_@\{e_d\} * e_d \hookrightarrow P * \forall e.\, (P \Rrightarrow \Box\,\Phi(x, v, e)) \end{array}\right\}$$

$$a\colon \mathsf{str}\ [x]\ (v + r - r)$$

$$\left\{\exists e.\ \begin{array}{l} \mathsf{PoPred}(e) * \mathsf{pc} \overset{\mathrm{r}}{\mapsto} (a+4)@\_ * \\ \mathsf{LastLocalWrite}(a, e) * \\ r \overset{\mathrm{r}}{\mapsto} \_@\{e_d\} \end{array}\right\}_{tid,\Phi}$$

Fig. 16. Proof rules for the instructions in the left thread of LB+artificialdata+data

5.4 Example: MP+rel+addr

a: str $[data]$ 42	c: r_1 := ldr $[flag]$
b: str$_{\mathrm{rel}}$ $[flag]$ 1	d: r_2 := ldr $[data + r_1 - r_1]$

Fig. 17. MP+rel+addr: $r_1 = 1 \Rightarrow r_2 = 42$

We give an example illustrating more complex reasoning about the memory event graph, on a message passing litmus test with the receiving thread ordered by an (artificial) address dependency. The example has two threads: one sending, and one receiving. The sending thread writes a value (in this case 42) to a data address in order to transfer it between threads, then writes 1 to a flag address to indicate that the data write has been completed. The flag write is a release write to ensure that the two writes are suitably ordered (this can also be achieved in other ways, for example with a dmb st). The receiving thread reads from the flag address to check whether a message has been passed to it, then reads from the data location. These two reads are ordered by using the result of the flag read to compute the address of the data read, resulting in an address dependency between the two reads. In this example, the address dependency is artificial, but similar shapes arise naturally when a message-passing idiom is used to transfer a pointer to a data structure between threads.

Specification. We want to be able to prove that if the load of the flag reads 1, then the load of the data will read 42; formally, $r_1 \overset{\mathrm{r}}{\mapsto} v * r_2 \overset{\mathrm{r}}{\mapsto} v' * d \hookrightarrow (v = 1 \longrightarrow v' = 42)$. The key statement of this specification is tied to an event, so it is only possible to use resources gained via message passing on events ob-after the receiving load.

We first specify the protocol used in the proof. For the data address, we pick $\Phi(\mathit{data}, v, e) \triangleq \mathrm{Initial}(e) \vee v = 42$, where $\mathrm{Initial}(e)$ denotes that the event is an initial write that necessarily has value 0. It is not possible to require that v be 42 in all cases, because the initial write would then not satisfy the protocol. For the flag address, we pick

$$\Phi(\mathit{flag}, v, e) \triangleq \mathrm{Initial}(e) \vee \begin{pmatrix} v = 1 * \mathit{tid}(e) = 1 * e{:}\mathrm{W}_{\mathrm{rel}}\ \mathit{flag}\ 1\ * \\ \exists e'.\ e'{:}\mathrm{W}\ \ \mathit{data}\ 42 * (e', e) \in \mathrm{po} \end{pmatrix}$$

requiring that a non-initial write to the flag address is only allowed if it is a release write of value 1, on the sending thread (which we assign *tid* 1), which is po after a write of 42 to the data address. Any other mechanism for generating ob ordering between the two writes would work, but for simplicity we specialise to using a release write.

With the protocol in hand, we can give a proof sketch for each thread. On the sending thread, we are required to show first that the data write a satisfies the protocol on *data*, which we can do straightforwardly because the right branch of the protocol only requires that the write have value 42. At the write of the data, we learn $\mathrm{PoPred}(a)$ and $a{:}\mathrm{W}\ \ \mathit{data}\ 42$. We are then required to show that the flag write satisfies the flag protocol. We can do so by instantiating the existential on the right hand side of the protocol with a, which is a write to *data* of 42, as required, and can be shown to be po-before the current event because it is the current po-predecessor.

On the receiving thread, we read from the flag, learning $r_1 \overset{\mathrm{r}}{\mapsto} v@\{c\}$ and $c \hookrightarrow \Phi(\mathit{flag}, v, b)$ for some b. Finally, we consider the data read. Because we have a data dependency, we have (c, d) in the data relation, which is in ob, and so we are able to use the resources tied to c. We learn $r_2 \overset{\mathrm{r}}{\mapsto} v'@\{d\}$ and $d \hookrightarrow \Phi(\mathit{data}, v', e)$ for some v' and e, and are required to prove $v = 1 \longrightarrow v' = 42$. Case splitting on $\Phi(\mathit{data}, v', e)$, we have $v' = 42$ immediately in the right case. In the left case, we derive a contradiction from $\mathrm{Initial}(e)$ and $v = 1$.

In detail, the contradiction is as follows: From $\Phi(\mathit{flag}, 1, b)$ we learn $b{:}\mathrm{W}_{\mathrm{rel}}\ \mathit{flag}\ 1$ and $a{:}\mathrm{W}\ \ \mathit{data}\ 42$ for some a such that $(a, b) \in \mathrm{po}$. Since e is an initial write to the same address as a, we know $(e, a) \in \mathrm{co}$, and since $(e, d) \in \mathrm{rf}$, we know $(d, a) \in \mathrm{fr}$ and therefore $(d, a) \in \mathrm{ob}$, since d and a are on different threads. Because b is a release write po-after a, we have $(a, b) \in \mathrm{ob}$; because c reads from b on a different thread, we have $(b, c) \in \mathrm{ob}$; and finally, because there is a data dependency between c and d, we have $(c, d) \in \mathrm{ob}$. By transitivity of ob, we obtain $(a, d) \in \mathrm{ob}$, and together with $(d, a) \in \mathrm{ob}$, we obtain $(d, d) \in \mathrm{ob}$, which contradicts the irreflexivity of ob.

While the only information we pass between threads here is the value of the data write, the protocol for the data address can carry any persistent resource (including arbitrary invariants) if they can be established at a, which would then be available tied to d.

5.5 Supporting Exclusives

Arm-A features atomic read-modify-write operations in two forms: atomic instructions (compare-and-swap, fetch-and-add, etc.), and the combination of load-exclusive/store-exclusive pairs. The rules we have described so far only support ‘plain’ loads and stores. We now explain how we can give strong rules for read-modify-writes that support transfer of non-duplicable resources: a load exclusive a of v from x should, if the subsequent store exclusive succeeds, give a non-duplicable resource P — not merely its duplicable portion $\Box\, P$.

Our solution relies on invariants, which we implement in §6.1 using the standard Iris construction combining higher-order ghost state and appropriate view shifts in the definition of weakest precondition [Jung et al. 2018]. Given a proposition P, $\boxed{P}$ is an invariant containing P, which is duplicable, and which can thus be transferred using our protocols. Using an invariant, we can then use the escrow pattern [Kaiser et al. 2017] to trade an exclusive token for the non-duplicable resource, with enough bookkeeping to capture the uniqueness of a successful read-modify-write on a given write. Therefore, the only change we need to make to support exclusives is merely to associate, in the definition of weakest precondition, an exclusive token ex(e) to each event e, and to make that token available to the rule for that event.

```
trylock($\ell$) $\triangleq$
  $r_1$ := ldr$_x$ [$\ell$]
  if ($r_1 \neq$ unlocked) return false
  $r_2$ := str$_x$ [$\ell$] locked
  dmb sy
  return ($r_2$ = success)
unlock($\ell$) $\triangleq$ str$_{rel}$ [$\ell$] unlocked
```

Fig. 18. Try-lock pseudocode

Using our proof rules, we prove the classical specification for simple try-lock (see Fig. 18), which we then use to prove a basic mutual exclusion example (in Fig. 19): if a writer takes the lock before writing to two variables, then a reader that takes the lock and reads from the two variables has to read the values before or after, but not a mixture.

```
if trylock($\ell$) {  ||  if trylock($\ell$) {
str [$x$] 1           ||  $r_1$ := ldr [$x$]
str [$y$] 1           ||  $r_2$ := ldr [$y$]
unlock($\ell$) }      ||  unlock($\ell$) }
```

Fig. 19. Example of mutual exclusion, with postcondition $r_1 = r_2$

5.6 Further Examples

To validate how AxSL works generally with the memory model of Arm-A, how it is likely to continue working with future changes, and how it could be ported to other memory models, we verify further examples that exercise different parts of the memory model. LB+dmbsy+data relies on the po; [dmb.full]; po clause of bob (see Fig. 4) to obtain the key lob edge on the left, but the proof is otherwise the same as for LB+artificialdata+data in §5.3. LB+ctrls relies on the ctrl; [W] clause of dob in both threads, but is otherwise the same. Similarly, MP+rel+dmbsy relies on bob in the right thread, and so does MP+rel+ctrl-isb via the more complex (ctrl|(addr; po)); [ISB]; po; [R] edge which appears incrementally, but their proofs are otherwise as in §5.4. To illustrate that AxSL can reason about communication between many threads, we verify an iterated version of MP, namely ISA2+rel+data+acq (Fig. 20) [Sarkar et al. 2011]; the proof is just an iterated version of the proof of MP. Finally, to illustrate that reasoning about coherence is still possible, albeit unpleasant (which we discuss further in §6.4), we verify two coherence tests: CoWW (Fig. 21) and CoRR (Fig. 22); the proofs work by symbolic execution followed by discarding the executions with cycles in co.

```
$a$: str [$x$] 42         ||  $c$: $r_1$ := ldr [$y$]  ||  $e$: $r_2$ := ldr [$z$]
$b$: str$_{rel}$ [$y$] 1  ||  $b$: str [$z$] $r_1$     ||  $d$: $r_3$ := ldr [$x$]
```

Fig. 20. ISA2+rel+data+acq: $r_2 = 1 \Rightarrow r_3 = 42$

6 MODEL, SOUNDNESS, AND ADEQUACY

In this section, we present our model of AxSL, which we define in §6.1 on top of the Iris base logic, and which is used to show soundness of the proof rules for the micro-instructions presented

$$\begin{array}{l} a\!: \mathsf{str}\ [x]\ 37 \\ b\!: \mathsf{str}\ [x]\ 42 \end{array}$$

Fig. 21. CoWW: $a \xrightarrow{\text{co}} b$

$$\begin{array}{l||l} a\!: \mathsf{str}\ [x]\ 42 & b\!: r_1 := \mathsf{ldr}\ [x] \\ & c\!: r_2 := \mathsf{ldr}\ [x] \end{array}$$

Fig. 22. CoRR: $r_1 = 42 \Longrightarrow r_2 = 42$

earlier. Afterwards, in §6.2, we prove the adequacy of AxSL (relying on the adequacy of the Iris base logic [Jung et al. 2018]). While we only provide an overview, this section does presuppose some knowledge of Iris. Finally, in §6.4, we discuss the challenge posed by coherence.

6.1 Semantics of Hoare Triples

State interpretation. As usual (and as briefly mentioned in §3), Iris ghost state is used to capture a logical interpretation of the physical semantics state, but in a way that can be fragmented into components of resources. The logical state interpretation LSI relates some logical state to the corresponding part of physical state for thread-local execution. It contains, among other things, the full view of a thread state's register map *regs* as $\boxed{\bullet regs}$, using the authoritative ghost state constructor of Iris [Jung et al. 2018]. Following usual Iris practice, the predicate $r \overset{\mathsf{r}}{\mapsto} rv$ (where rv is a v, E pair), which we use for local reasoning, is then defined as a fragmental view of *regs*: $\boxed{\circ[r \mapsto rv]}$. With these definitions, we obtain the following two key rules:

SI-REG-AGREE
$$\boxed{\bullet regs} * \boxed{\circ[r \mapsto rv]} \twoheadrightarrow\!\!\!* \; regs(r) = rv$$

SI-REG-UPDATE
$$\boxed{\bullet regs} * \boxed{\circ[r \mapsto rv]} \Rrightarrow \boxed{\bullet regs[r \mapsto rv']} * \boxed{\circ[r \mapsto rv']}$$

SI-REG-AGREE states that the two views (authoritative and fragmental) agree on the value of r, and rule SI-REG-UPDATE allows us to update them with a view shift.

For a complete thread state $T = \langle T.\mathsf{p}, T.\mathsf{r}\rangle$, the rest of $\mathsf{LSI}_{tid}(T.\mathsf{r})$ is defined in the same spirit, and in such a way that view shifts $\mathsf{LSI}_{tid}(T.\mathsf{r}) \Rrightarrow \mathsf{LSI}_{tid}(T'.\mathsf{r})$ (as found in the weakest precondition below) can mirror transitions from $T.\mathsf{r}$ to $T'.\mathsf{r}$ in the opax semantics.

While LSI_{tid} is a thread-local predicate which only involves assertions of thread *tid* (for Iris experts, thread-local assertions use distinct ghost names), we use the same idea to connect global states to global resources (both graph resources and tied-to assertions) in the weakest precondition definition below.

Hoare triples. Following many other Iris-based separation logics, we define the meaning of our Hoare triples for microinstruction program p using a notion of weakest precondition plus a thin abstraction layer:

$$\{P\}\ p\ \{Q\}_{tid,\Phi} = \Box(\forall T.\ (\mathsf{LSI}_{tid}(T.\mathsf{r}) * T.\mathsf{p} = p) \twoheadrightarrow\!\!\!* P \twoheadrightarrow\!\!\!* \mathsf{wp}\ (\mathsf{Ctd}\ T)\ \{T'.\ Q\}_{tid,\Phi}).$$

Here, the $\Box$ is used to enforce that Hoare triples are persistent (thus duplicable) knowledge. The rest of the definition is quantified over an ongoing local execution state T of the opax semantics. Intuitively, the weakest precondition $\mathsf{wp}\ (\mathsf{Ctd}\ T)\ \{T'.\ Q\}_{tid,\Phi}$ asserts that the opax state $\mathsf{Ctd}\ T$ can run the program $T.\mathsf{p}$ safely, and that, when the execution terminates with T', the post condition Q holds. When showing the weakest precondition, we get to assume not only the precondition P from the Hoare triple, but also that the microinstruction is indeed linked to the local execution state ($T.\mathsf{p} = p$), and that we have thread-local logical resource $\mathsf{LSI}_{tid}(T.\mathsf{r})$ interpreting $T.\mathsf{r}$.

Weakest preconditions. We now turn to the definition of the weakest precondition predicate. There are two aspects of its definition which are somewhat non-standard compared to other Iris-based definitions of weakest precondition predicates: the first is that it maintains consistency between the execution of the thread in the opax semantics and the global execution graph, to ensure sound graph reasoning; the second is that it keeps track of all tied-to assertions, to enable sound transfer of tied resources between events (this latter aspect is somewhat reminiscent of how invariants are tracked in the weakest precondition for the standard Iris program logic [Jung et al. 2018]). We do not include the full formal definition, but explain the key ideas of the definition:

$$\begin{array}{l}
\mathsf{wp}\ s\ \{Q\}_{tid,\Phi} \approx \\
\quad \text{if } s = \mathsf{Done}\ \langle\_\rangle \text{ then } \mathsf{PullOutTied}(tid, Q) \\
\quad \text{else } \forall T, e, X, I. \left(\begin{array}{l}
s = \mathsf{Ctd}\ T \mathrel{-\!\!*} e = T.\mathsf{e} \mathrel{-\!\!*} \mathsf{Valid}(X) \mathrel{-\!\!*} \\
(\Box\, \mathsf{SI}_{\mathsf{G}}(X, I)) \mathrel{-\!\!*} \cdots \mathrel{-\!\!*} \forall s', T'.\, s \xrightarrow{tid,X,I}_{\mathsf{h}} s' \mathrel{-\!\!*} \\
\left(\forall \sigma.\, \mathsf{SI}_{\mathsf{T}}(\sigma) \Rrightarrow \exists \sigma'.\, \left(\begin{array}{l}
\mathsf{FlowImp}_{X,\Phi}(e, \sigma, \sigma') * \mathsf{SI}_{\mathsf{T}}(\sigma') * \\
\cdots * \mathsf{LSI}_{tid}(T'.\mathsf{r}) * \mathsf{wp}\ s'\ \{Q\}_{tid,\Phi}
\end{array}\right)\right)
\end{array}\right) \\
\mathsf{PullOutTied}(tid, Q) \approx \forall \sigma.\, \mathsf{SI}_{\mathsf{T}}(\sigma) \mathrel{-\!\!*} \left(\mathsf{SI}_{\mathsf{T}}(\sigma) * \left(\mathop{\ast}_{\{R \mid (e \mapsto R) \in \sigma \land \mathsf{Local}(tid, e)\}} R \Rrightarrow Q\right)\right)
\end{array}$$

The definition makes a case distinction on whether the thread *tid* terminates with opax state s.

In the terminating case, when $s = \mathsf{Done}\ \langle\_\rangle$, we basically require that postcondition Q holds, modulo the unusual PullOutTied predicate, which we we will elaborate on below. In the other case, if the thread can take a step in the opax semantics and reach a new s', then the weakest precondition should hold recursively for s' — the weakest precondition predicate is defined as the least fixed point satisfying the recursive equation. Our definition does not require that s can take a step in that case; thus, our definition does not enforce progress.

Besides the local state interpretation LSI_{tid}, which we have already seen above, the definition contains two global interpretations SI_{T} and SI_{G} for the full authoritative view of all tied assertions σ (σ is a logical map from event IDs to propositions) and the shared execution graph X with the instruction memory I, respectively. The predicate SI_{T} is defined in such a way that it, together with the fragmental tied-to-assertions, enjoys similar agreement and update rules as for the register map shown above (the rules are slightly different since σ is higher-order, in the sense that it maps to predicates). In contrast, the predicate SI_{G} is persistent (decorated with $\Box$), and then there are no update rules for it, due to the immutability of the graph and of the instruction memory.

The first line of the definition deals with a terminated thread, in which case we have to show $\mathsf{PullOutTied}(tid, Q)$, which requires us to establish the postcondition Q, assuming that the predicates pulled out from local tied assertions hold — we have already seen an example of how this is used, namely in the final reasoning step (in each thread) in Fig. 14. Technically, the definition of $\mathsf{PullOutTied}(tid, Q)$ makes use of the agreement rule for a local event e, which roughly says that $\mathsf{SI}_{\mathsf{T}}(\sigma) * e \hookrightarrow R$ implies that $e \mapsto R$ is in σ and thus that R holds (this is an approximate description, the formal details are a bit more subtle because of the higher-order nature of the σ map mentioned above).

The step case universally quantifies over the local state T of $s = \mathsf{Ctd}\ T$, the event ID of the event e associated with the head microinstruction (equals to $T.\mathsf{e}$), and a global execution graph X, which is assumed to be valid (i.e., well formed and consistent) as expressed by the Valid predicate. The reduction step $s \xrightarrow{tid,X,I}_{\mathsf{h}} s'$ of the opax semantics on the third line ensures that X is aligned with the foremost micro-instruction of $T.\mathsf{p}$. Then, in the last line, after a view shift to reflect the state transition, the weakest precondition has to hold recursively for s' and, moreover, the thread-local logical resources predicate $\mathsf{LSI}_{tid}(T'.\mathsf{r})$ also has to hold. Finally, the fourth line is a view shift, which essentially allows us to update a fragment of σ (to σ') and the associated tied assertions. Crucially, for the sake of the soundness proof, the update has to follow the flow implication FlowImp, which we explain below.

Flow implication. The $\mathsf{FlowImp}_{X,\Phi}(e, \sigma, \sigma')$ predicate regulates the flow and update of resources that may happen at memory event e. (The flow implications we have seen in proof rules earlier are instances of this one.) Intuitively, the rule expresses that the sum of resources pushed to e along its incoming ob edges implies (with a view shift) the resources given out along outgoing ob edges, plus the leftovers tied to e. In more detail, we define the predicate as follows:

$$\begin{array}{l}
\mathsf{FlowImp}_{X,\Phi}(e, \sigma, \sigma') \triangleq \\
\quad \exists \sigma_{in}, \sigma_{res}, R.\, \mathsf{Detach}_X(\sigma, \sigma_{in}, e) = \sigma_{res} * \\
\quad \Big(\big(\mathop{\ast}_{(e \mapsto R_{in}) \in \sigma_{in}} R_{in} \big) * \big(\mathop{\ast}_{e_{\mathrm{obs}} \in \mathsf{ObsPred}(e,X)} \square\, \mathsf{GiveRes}(e_{\mathrm{obs}}, \Phi) \big) \Rrightarrow R * \square\, \mathsf{GiveRes}(e, \Phi) \Big) * \\
\quad \sigma' = \sigma_{res}[e \mapsto R]
\end{array}$$

We divide the update from resource map σ to resource map σ' into a sequence of three actions: detach, exchange, and tie. The first line captures the detachment: σ is split into σ_{in} and σ_{res}, where σ_{in} is a fragment of σ which represents the resources *detached* from the lob predecessors of e, as determined by the user-provided tied assertions and σ_{res} is the remaining global map, after detaching σ_{in}. The second line does the resource exchange, using a view shift. Given the resources of σ_{in}, one may update and *exchange* resources with other threads. Specifically, persistent resource specified by $\square\, \mathsf{GiveRes}(e_{\mathrm{obs}}, \Phi)$ may flow from obs predecessors e_{obs} to e, and $\square\, \mathsf{GiveRes}(e, \Phi)$ has to be given away to potential obs successors. $\mathsf{GiveRes}(e_{\mathrm{obs}}, \Phi)$ is essentially defined as the protocol resource Φ on e, plus a thin layer to resolve the type mismatch. The third line does the tying: the remaining resource R is *tied* to e, by extending the map σ_{res}.

Supporting framing and invariants. Recall that splitting of tied assertions is useful for proving examples (cf. Fig. 8). Modeling such splitting is, however, quite challenging due to its higher-order nature. We addressed this challenge by first implementing a splitting with the help of the interpretation SI_{T}: $e \hookrightarrow (P * Q) \mathrel{-\!\!*} (\forall \sigma.\, \mathsf{SI}_{\mathsf{T}}(\sigma) \Rrightarrow (\mathsf{SI}_{\mathsf{T}}(\sigma) * e \hookrightarrow P * e \hookrightarrow Q)))$. Here, the view shift allows us to update tied assertions and the interpretation without changing the value of the global map σ. Then, the view shift structure is hidden by tweaking the definition of the weakest precondition to close it under this pattern.

Supporting invariants, on the other hand, only requires minor updates to the weakest precondition definition: we just replace the plain view shift in FlowImp with an Iris built-in variant (a combination of the later and the fancy update modality) that allows opening and closing of invariants, similar to [Jung et al. 2018].

Infinite executions. Because of the open problem with infinite executions in the memory model, our definition of weakest precondition in §6.1 does not take measures to handle infinite executions either, and is defined using a least fixpoint for ease of definition (as described, it still uses the later operator for higher-order ghost state).

Soundness. Using the model, we can now prove soundness of the proof rules for microinstructions:

Theorem 6.1. *The AxSL proof rules for microinstructions are sound.*

6.2 Adequacy

The soundness of AxSL is expressed by the following meta-level adequacy theorem, which combines valid program executions and thread-local proofs in AxSL, and shows that the conjunction of the threads' postconditions is valid at the meta level:

Theorem 6.2 (Adequacy). *For initial thread states $\vec{s}$, meta-level propositions $\vec{P}$ (one for each thread), and valid execution graph X, we have*

$$\left(\bigwedge_{tid=1}^{n} \vec{s}(tid) \xrightarrow{tid,X,I}{}_{\mathsf{h}}^{*} \mathsf{Done}\ \_\right) \Rightarrow$$
$$\left(\exists \Phi.\ \vdash (\Box\, \mathsf{InitRes}(\Phi)) * \ldots * (\Box\, \mathsf{SI}_{\mathsf{G}}(X,I)) * \mathop{\ast}_{tid=1}^{n} \left(\ldots * \mathsf{wp}\ \vec{s}(tid)\ \left\{\left\lceil \vec{P}(tid)\right\rceil\right\}_{tid,\Phi}\right)\right) \Rightarrow$$
$$\left(\bigwedge_{tid=1}^{n} \vec{P}(tid)\right)$$

Here $\vec{s}$ is a sequence of initial thread states (one for each thread). The first hypothesis ensures that the memory graph X reflects the behaviours of a complete program by assuming terminating executions of all threads starting from $\vec{s}$. The second hypothesis assumes that we have proofs in AxSL of weakest preconditions, one for each thread, using the same protocol Φ. This is where we require that the *same* protocol Φ is agreed upon between the thread-local proofs of the weakest preconditions for each thread, as well as agreement about the execution graph X and instruction memory I via the global state interpretation (the rest of the state interpretation requires some further plumbing, which we elide). The post conditions are assumed to be the meta-level propositions $\vec{P}$, lifted to AxSL by $\lceil \_ \rceil$. (This lifting is similar to what happens in other Iris-based program logics: it simply embeds a proposition P from the meta-level as the corresponding proposition in AxSL.) From these, adequacy tells us that $\vec{P}$ hold in the meta-logic as well. The importance of this adequacy theorem lies in the fact that is means that we do not have to trust or even understand all the intricacies of AxSL — once we have proved weakest preconditions for each thread using the AxSL proof rules, then the theorem guarantees that the postconditions $\vec{P}$ do indeed hold at the meta-level (assuming each thread's execution terminates).

Overview of the proof. The crux of the adequacy proof is to compose thread-local reasoning results, specified as weakest preconditions. For classic concurrent separation logics (including most Iris-based program logics using the standard weakest precondition construction, including iGPS), the proof of adequacy works by induction on the program execution trace. This is possible in their setups because resources are annotated on program points (or, interchangeably, on po edges), and their weakest preconditions, from our perspective, essentially use flow implications to flow resources between program points in program order *on the fly.*

In AxSL, resources are used to annotate ob, but the weakest precondition still has to collect the flow implications in po. This mismatch poses the main challenge to the proof of adequacy, since one cannot do induction on the potentially cyclic po ∪ ob. We resolve this tension by *stratifying* the proof into two phases: phase one collects flow implications along po, and phase two applies them along ob. In the rest of the section, we sketch the two phases in more detail, and leave the discussion on the relation to RSL/FSL soundness proofs to related work.

Phase one. Roughly, phase one constructs a valid annotation on every ob edge of the execution graph by collecting and chaining flow implications from all events. The edge annotation records the history of how resources evolve and are transferred soundly along ob in a complete program execution, as inspired by RSL/FSL. Thanks to our definition of weakest precondition, the local edge annotations for an event, which is essentially a resource transformer, can be easily derived from the corresponding tied-to map update from σ to σ' specified by the flow implication FlowImp. Furthermore, the fact that every such update follows the flow implication guarantees that these local annotations can be composed both vertically (in po) and horizontally (between threads) to get a global edge annotation, as follows.

In each thread, the vertical composition is done by induction on the thread's trace to unfold the recursively defined weakest precondition. This allows us to collect local annotations along po to obtain an annotation for the thread. Next, the derived annotations are glued horizontally (between threads) to obtain a global annotation for ob edges, which is possible since all resource exchange across threads (as annotated on obs) are specified by the same rely-guarantee protocol Φ.

Phase two. Given the edge annotation from phase one, phase two stitches the flow implications by induction on ob. For the base case of the induction, we require the user to show that the resources specified by Φ for the initial events of all memory locations can be established. The allocated resources are then used as the starting point for the application of the flow implications along (an order extension of) ob. The resources at the end of the process are the sum of those tied to all events, which we can combine to show the postconditions. Finally, we apply the soundness result of the Iris base logic to conclude that $\vec{P}$ holds in the meta-logic.

6.3 Proof Effort

Much of the effort was in the overall design of the logic to overcome the challenge to soundness posed by load buffering. Shaping the idea to fit Iris, and detailing the model of assertions and the definition of weakest preconditions, took over a person-year, but the result has been robust to small changes, for example to add exclusives. The adequacy theorem has the most significant proof: it took two or three person-months to mechanise after initial design work, and it stands around a thousand lines of Coq. Writing and proving an instruction proof rule takes about half a person-day now that we have enough of them that flesh out a pattern. Finding an overall proof structure for a new shape of litmus test takes a few person-days; in fact, it is very similar to what is needed for example in RSL. Adapting a proof from one variant of a litmus test to another is just a few hours' work, and less than a hundred lines of Coq. Overall, the mechanisation is divided as follows:

Item	**LoC**
Prelude (incl. outcome interface and infrastructure)	~6000
Language definition and lemmas	~1500
Definition of axiomatic model and lemmas	~3500
Iris CMRAs	~800
WPs and lemmas	~4000
Proof rules and their soundness proofs	~3000
Adequacy	~1000
Examples	~1500

6.4 Coherence

The memory model of Arm-A involves two main axioms: `external`, which requires ordered-before to be acyclic, and `internal`, which requires `po-loc|ca|rf` to be acyclic, which effectively enforces per-location sequential consistency (the atomicity axiom is much more 'local' and easier to use, as per §5.5). This latter order is sometimes also called coherence, or (to avoid confusion with the co relation, which is merely part of it) extended coherence.

The way AxSL is defined in Iris above the opax semantics using the Fig. 4 memory model means it captures both axioms. However, it focuses on the external axiom, and leverages the acyclicity of ob to allow sound transfer of resources along ob. This means that AxSL cannot soundly allow transfer resources along the potentially cyclic combination of ob and extended coherence. It is always possible to reason about extended coherence by brute force in AxSL, by explicit graph reasoning, but this is unsatisfactory. This is a definite limitation of AxSL, as many common programming and reasoning idioms rely on reasoning about coherence, in particular the notion of non-atomics.

7 RELATED WORK

The Lace logic [Bornat et al. 2015] shares the same core idea of explicitly tracking ordering between memory events (which they call 'laces'), and of flowing assertions along edges (which they call 'embroidery') of an axiomatic memory model. Their setup looks quite different on the surface, as their approach to ordering is top-down (in the style of Crary and Sullivan [Crary and Sullivan 2015]), stating which instructions they require ordering from, rather than our bottom-up approach, in which we infer order from the instructions of the program. The main difference is that they try to talk about variables (memory locations), and so, to soundly flow assertions along edges, they need to check for interference on said variables, which is a whole-program check. In addition, they leave supporting separation as future work, and thus feature no notion of transfer of resources. However, Lace features modalities to ease reasoning about coherence, whereas it has to be done by graph reasoning in our logic. Lace was implemented using a custom proof checker, with no proof of its soundness.

The Ogre and Pythia invariance proof method [Alglave and Cousot 2017] refines Owicki-Gries without auxiliary variables (which are unsound for relaxed memory [Lahav and Vafeiadis 2015]) by working with memory events (via program counters), and their "pythia" variables keep track of the values of reads. Their method is parameterised by an axiomatic memory model expressed by relational algebra in the `.cat` format [Alglave et al. 2014]. They show that their proof method is sound and relatively complete, but their invariants are whole-program, and they leave development of abstractions that make proofs tractable as future work.

Our flow implications are inspired by those of RSL and FSL. However, thanks to the phrasing of the memory model of Arm-A, we give a single, generic definition, instead of one based on case analysis of instructions. In addition, the pervasive effect of undefined behaviour (stemming from data races on non-atomics and uninitialised reads) in C11, substantially complicates the definition of flow implications of RSL and FSL.

The proofs of adequacy of RSL and FSL are somewhat similar to ours, being also based on chaining flow implications along a global acyclic synchronisation relation, C11's *happens-before* (hb). However, the memory model of C11 is substantially different from that of Arm, and in particular, despite a similar role, hb is substantially different from ob: it is defined as $\mathsf{hb} \triangleq (\mathsf{sb} \cup \mathsf{sw})^+$, where sb is C11's counterpart to po, and sw is C11's loose counterpart to obs. This allows them to freely persist resources along program order, and so their flow implications refer to immediate program order successors and predecessors of instructions, which means that they have their postcondition immediately in hand, and do not need to collect tied resources. It also means that, unlike ours, their proof of adequacy can be along program order.

FSL [Doko and Vafeiadis 2016] extends RSL to make it possible to transfer resources using C11 'relaxed' access that are suitably fenced by guarding resources with modalities: a relaxed load, which imposes little order by itself, merely obtains ∇P, which is not usable by itself, but which an acquire fence turns into P. Symmetrically, P itself cannot be sent away by a mere relaxed store, but a release fence turns P into ΔP, which a relaxed store can send away. Our $a \hookrightarrow P$ assertion can be seen as an indexed version of ∇P, keeping track of the source of the assertion in a way that is compatible with ghost state, even with cycles in $\mathsf{po} \cup \mathsf{rf}$.

SLR [Svendsen et al. 2018] targets the Promising Semantics [Kang et al. 2017] designed to fix the out-of-thin-air problem of C11. SLR takes advantage of the extra strength to enable coherence (sc-per-location) reasoning on relaxed accesses, but does not allow resource transfer using relaxed accesses. SLR features an assertion to keep track of writes that is somewhat similar to our NoLocalWrites and LastLocalWrite assertions: $W^{\pi}(x, X)$ imposes a lower bound X on the set of writes done so far to location x; however, they use it for coherence reasoning, rather than to bound internal reads.

Compass [Dang et al. 2022] is a specification framework for the ORC11 memory model (which forbids load buffering) that gives programs specifications in terms of event graphs the inter-thread edges of which are induced by data structure operations, for example from an enqueue to the dequeue of the same value, which generalises rf.

We conjecture that the unpublished extension of ribbon proofs to relaxed memory mentioned in [Wickerson et al. 2013] would have had some similarities to our setup, with unclosed ribbons standing for tied assertions, and flow implications imposing conditions on when ribbons can be joined.

8 CONCLUSION

The very relaxed concurrency memory models of the Arm-A hardware architecture, in which execution order (ob) does not follow program order, makes syntax-directed and thread-modular reasoning challenging. The need to program it directly for performance and for access to systems features makes this challenge unavoidable. Our logic, AxSL, addresses this challenge through assertions that track how ob is induced by the program text, and makes reasoning tractable by flowing higher-order ghost state along ob. This allows us to capture and thus validate key synchronisation idioms of Arm-A. Moreover, our approach relies essentially just on the acyclicity of ob, and should therefore generalise to similar hardware architectures.

This opens up a potential approach for reasoning about a wide range of axiomatic models, and there are many important extensions to explore, e.g. to integrate with the full Arm-A or RISC-V ISAs, to cover mixed-size accesses, and to cover systems features including instruction-fetch and virtual memory. An important challenge is to tackle reasoning involving not only ob but also coherence, in particular as leveraged by non-atomics, even though their union can have cycles.

A FURTHER PROOF RULES

A.1 MemWrite **Rule**

HT-MICRO-MEMWRITE

$$\left\{\begin{array}{l} (1)(\mathsf{LastLocalWrite}(x, v_{\mathrm{last}}) \vee \mathsf{NoLocalWrites}(x)) * d = (\mathrm{dom}(regs), \emptyset) * \\ \mathsf{PoPred}(e_{\mathrm{po}}) * \mathsf{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\ \displaystyle\mathop{\Huge *}_{(r \mapsto v@E) \in regs} r \overset{\mathrm{r}}{\hookrightarrow} v@E * \mathop{\Huge *}_{(e_{\mathrm{lob}} \mapsto P_{\mathrm{lob}}) \in m} (e_{\mathrm{lob}} \rightsquigarrow P_{\mathrm{lob}}) * \\ \forall e. \left(\begin{array}{l} (2)\mathsf{GraphFacts}'(os, vr, x, v, e, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) \mathrel{-\!\!*} \\ (\mathsf{Lob}(\mathrm{dom}(m), e) * (3)\mathsf{Flow}'_{\Phi}(e, x, v, m, P)) \end{array} \right) \end{array}\right\}$$

$$\mathsf{MemWrite}\ os\ vr\ x\ v\ d$$

$$\left\{ (), E. \begin{array}{l} \exists e. E = \{e\} * (4)\mathsf{LastLocalWrite}(x, v) * \mathsf{PoPred}(e) * \mathsf{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\ (5) \displaystyle\mathop{\Huge *}_{(r \mapsto v@E) \in regs} r \overset{\mathrm{r}}{\hookrightarrow} v@E * (6)\mathsf{GraphFacts}'(os, vr, x, v, e, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) * \\ (7) e \rightsquigarrow P(x, v) \end{array} \right\}_{tid, \Phi}$$

Fig. 23. A proof rule of AxSL for the MemWrite microinstruction. As for the MemRead rule in Fig. 12 the user provides m, a thread-local map from events to the resources consumed.

Perhaps surprisingly, the basic rule for MemWrite is extremely similar to that for MemRead given in Fig. 12, so we only highlight the key differences here. Unlike the read rule, the write rule is not affected by the presence of previous local writes, and so can take either a NoLocalWrites or a LastLocalWrite resource in (1). Because a new local write is produced, whichever resource is passed in (1), a LastLocalWrite carrying the new event is returned (4). The definition of GraphFacts′ at (2) and (6) differs from the version used in Fig. 12 because write events induce a different collection of incoming edges, most importantly lacking an incoming rf edge. Similarly the definition of Flow′ (3) is changed from showing the incoming tied resources and protocol imply the new tied resource to showing the incoming tied resources imply the new tied resource and the outgoing protocol. Finally we note that the produced tied resource (7) is parameterised only on the value written and the identifier of the new node, not on the identifier of some write event being read from.

A.2 MemRead Rule with Local Writes

HT-MICRO-MEMREAD-RDEP-EXT-LOCAL

$$\left\{\begin{array}{l} (1)\mathsf{LastLocalWrite}(x, v') * d = (\mathsf{dom}(regs), \emptyset) * \\ \mathsf{PoPred}(e_{\mathrm{po}}) * \mathsf{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\ \displaystyle\mathop{\Huge *}_{(r \mapsto v@E) \in regs} r \overset{\mathrm{r}}{\hookrightarrow} v@E * \mathop{\Huge *}_{(e_{\mathrm{lob}} \mapsto P_{\mathrm{lob}}) \in m} (e \looparrowright P_{\mathrm{lob}}) * \\ \forall e, v, e_w. \left(\begin{array}{l} \mathsf{GraphFacts}(e, os, vr, x, v, e_w, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) \mathrel{-\!\!*} \\ (\mathsf{Lob}(\mathsf{dom}(m), e) * \mathsf{Flow}_{\Phi}(e, x, v, e_w, m, P)) \end{array} \right) \end{array}\right\}$$

$$\mathsf{MemRead}\ os\ vr\ x\ d$$

$$\left\{ v, E. \begin{array}{l} \exists e, e_w.\, E = \{e\} * (2)\mathsf{LastLocalWrite}(x, v') * \mathsf{PoPred}(e) * \mathsf{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\ \mathsf{GraphFacts}(e, os, vr, x, v, e_w, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) * \displaystyle\mathop{\Huge *}_{(r \mapsto v@E) \in regs} r \overset{\mathrm{r}}{\hookrightarrow} v@E * \\ (3)((e \looparrowright P(x, v, e_w)) \vee v = v') \end{array} \right\}_{tid, \Phi}$$

Fig. 24. A proof rule of AxSL for the MemRead microinstruction, similar to that shown in Fig. 12, but specialised to handle the case where the current thread contains a previous local write to the address being read from.

Fig. 24 shows a variant of the MemRead rule in Fig. 12 which allows for a previous local write to the address being read from. The prerequisites of the rule differ only in that it requires LastLocalWrite (1) instead of NoLocalWrites, which is still returned unchanged (2). The conclusion becomes a disjunction (3), if an external write is read from then the rule produces the expected tied resource. However if the local write is read from no resource transfer takes place, because thread local reads do not provide ob ordering, so we learn only that the value read is equal to the value written in the most recent local write. While this does not provide as powerful a mechanism for resource transfer as might be desired, we do expect it to allow some reasoning since in synchronisation primitives it will generally be necessary for an external thread to write distinct values to those written by the local thread.

ACKNOWLEDGMENTS

We thank Amin Timany for helpful discussions concerning framing.

This work was supported in part by Google, through ASPIRE faculty awards and other funding to Birkedal, Pichon-Pharabod, and Sewell; in part by Arm (Sewell); by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 789108, AdG ELVER, Sewell); by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation (Birkedal); by an AUFF starter grant (Pichon-Pharabod); and by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694.

REFERENCES

Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 3–18. https://doi.org/10.1145/3009837.3009883

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. https://doi.org/10.1145/3458926

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 405–418. https://doi.org/10.1145/3173162.3177156

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 258–272. https://doi.org/10.1007/978-3-642-14295-6_25

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. https://doi.org/10.1145/2627752

Arm Ltd. 2023. *ARM Architecture Reference Manual (for A-profile architecture).* Arm Ltd. ARM DDI 0487J.a (ID042523), https://developer.arm.com/documentation/ddi0487/latest/, Accessed 2023-07-04.

Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 303–316. https://doi.org/10.1007/978-3-030-81685-8_14

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *ESOP.* 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 509–520. https://doi.org/10.1145/2103656.2103717

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. https://doi.org/10.1145/1926385.1926394

P. Becker (Ed.). 2011. *Programming Languages — C++.* ISO/IEC 14882:2011. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 68–78. https://doi.org/10.1145/1375581.1375591

Richard Bornat, Jade Alglave, and Matthew J. Parkinson. 2015. New Lace and Arsenic: adventures in weak memory with a program logic. *CoRR* abs/1512.01416 (2015). arXiv:1512.01416 http://arxiv.org/abs/1512.01416

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *ACM SIGLOG News* 3, 3 (2016), 47–65. https://doi.org/10.1145/2984450.2984457

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383

Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential reasoning for optimizing compilers under weak memory concurrency. 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, 213–228. https://doi.org/10.1145/3519939.3523718

Karl Crary and Michael J. Sullivan. 2015. A Calculus for Relaxed Memory. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 623–636. https://doi.org/10.1145/2676726.2676984

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. 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, 792–808. https://doi.org/10.1145/3519939.3523451

Will Deacon. 2016. The ARMv8 Application Level Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 287–300. https://doi.org/10.1145/2429069.2429104

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Marko Doko. 2021. *Program Logic for Weak Memory Concurrency.* Ph. D. Dissertation. Kaiserslautern University of Technology, Germany. https://kluedo.ub.rptu.de/frontdoor/index/index/docId/6679

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration verification across software and hardware for a simple embedded system. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 604–619. https://doi.org/10.1145/3453483.3454065

Robert W. Floyd. 1967. Assigning Meanings to Programs. In *Proceedings of the Symposium in Applied Mathematics*, Vol. 19. American Mathematical Society, 19–32.

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 *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 429–442. https://doi.org/10.1145/3009837.3009839

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

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, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, Milos Prvulovic (Ed.). ACM, 635–646. https://doi.org/10.1145/2830772.2830775

Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. 2023. Research data for An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic. https://github.com/logsem/AxSL, https://doi.org/10.17863/CAM.104080. https://doi.org/10.17863/CAM.104080

Lisa Higham, LillAnne Jackson, and Jalal Kawash. 2007. Specifying memory consistency of write buffer multiprocessors. *ACM Trans. Comput. Syst.* 25, 1 (2007), 1. https://doi.org/10.1145/1189736.1189737

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 759–767. https://doi.org/10.1145/2933575.2934536

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. https://doi.org/10.1145/3009837.3009850

Prince Kohli, Gil Neiger, and Mustaque Ahamad. 1993. A Characterization of Scalable Shared Memories. In *Proceedings of the 1993 International Conference on Parallel Processing, Syracuse University, NY, USA, August 16-20, 1993. Volume I: Architecture*, C. Y. Roger Chen and P. Bruce Berra (Eds.). CRC Press, 332–335. https://doi.org/10.1109/ICPP.1993.15

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9135)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Software Eng.* 3, 2 (1977), 125–143. https://doi.org/10.1109/TSE.1977.229904

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 362–376. https://doi.org/10.1145/3385412.3386010

Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2023, Orlando, Florida, June 17-21, 2023.* ACM, 1438–1462. https://doi.org/10.1145/3591279

Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2016. Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings. *CoRR* abs/1611.01507 (2016). arXiv:1611.01507 http://arxiv.org/abs/1611.01507

Paul E. McKenney, Ulrich Weigand, Andrea Parri, Boqun Feng, and Alan Stern. 2020. Linux-Kernel Memory Model. ISO/IEC JTC1 SC22 WG21 P0124R7 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0124r7.html.

F.L. Morris and C.B. Jones. 1984. An Early Program Proof by Alan Turing. *Annals of the History of Computing* 6, 2 (1984), 139–143. https://doi.org/10.1109/MAHC.1984.10017

Magnus O. Myreen. 2009. *Formal verification of machine-code programs.* Ph. D. Dissertation. University of Cambridge.

Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. 2007. Hoare Logic for ARM Machine Code. In *Fundamentals of Software Engineering (FSEN)*, Farhad Arbab and Marjan Sirjani (Eds.). Springer, 272–286.

Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare Logic for Realistically Modelled Machine Code. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Orna Grumberg and Michael Huth (Eds.). Springer, 568–582.

Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. 2008. Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic. In *Formal Methods in Computer-Aided Design (FMCAD)*, Alessandro Cimatti and Robert B. Jones (Eds.). IEEE, 1–8.

Peter Naur. 1966. Proofs of algorithms by general snapshots. *BIT* 6, 310–316.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2142)*, Laurent Fribourg (Ed.). Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6 (1976), 319–340. https://doi.org/10.1007/BF00268134

Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 622–633. https://doi.org/10.1145/2837614.2837616

Christopher Pulte. 2018. *The Semantics of Multicopy Atomic ARMv8 and RISC-V.* Ph. D. Dissertation. University of Cambridge, UK. https://doi.org/10.17863/CAM.39379

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. https://doi.org/10.1145/3158107

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. https://doi.org/10.1145/3314221.3314624

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

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. https://doi.org/10.1145/3519939.3523434

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 311–322. https://doi.org/10.1145/2254064.2254102

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 175–186. https://doi.org/10.1145/1993498.1993520

Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 379–391. https://doi.org/10.1145/1480881.1480929

Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 143–173. https://doi.org/10.1007/978-3-030-99336-8_6

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. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 626–655. https://doi.org/10.1007/978-3-030-44914-8_23

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 866–881. https://doi.org/10.1145/3477132.3483560

Alan M. Turing. 1949. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*. Mathematical Laboratory, Cambridge, UK, 67–69. https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-8

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 691–707. https://doi.org/10.1145/2660193.2660243

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 867–884. https://doi.org/10.1145/2509136.2509532

Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213*. Technical Report. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

John Wickerson, Mike Dodds, and Matthew J. Parkinson. 2013. Ribbon Proofs for Separation Logic. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 189–208. https://doi.org/10.1007/978-3-642-37036-6_12

Received 2023-07-11; accepted 2023-11-07