An Axiomatic Basis for Computer Programming on Relaxed Hardware Architectures: The AxSL Logics

ZONGYUAN LIU, Aarhus University, Denmark
ANGUS HAMMOND, University of Cambridge, UK
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 family of separation logics for relaxed hardware memory models, and instantiate it on sequential consistency and on the Arm-A memory model. The Arm-A instance captures the fine-grained reasoning underpinning the low-overhead synchronisation idioms used by high-performance systems code. 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.

By instantiating AxSL on different memory models, we demonstrate the generality of our approach, and show that it is largely generic in the axiomatic model and in the instruction-set semantics, offering a potential way forward for compositional reasoning for other 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:
Zongyuan Liu, Angus Hammond, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. 2024. An Axiomatic Basis for Computer Programming on Relaxed Hardware Architectures: The AxSL Logics. 1, 1 (October 2024), 62 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

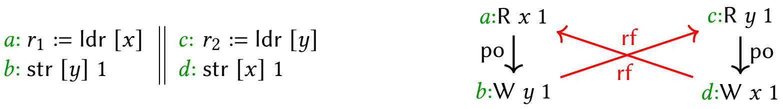
Authors' addresses: Zongyuan Liu, zy.liu@cs.au.dk, Aarhus University, Åbogade 34, Aarhus, Denmark, 8200; Angus Hammond, Angus.Hammond@cl.cam.ac.uk, University of Cambridge, Computer Laboratory, JJ Thomson Avenue, Cambridge, UK, CB3 0FD; Thibaut Pérami, Thibaut.Perami@cl.cam.ac.uk, University of Cambridge, 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.

Manuscript submitted to ACM

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? For example, 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, nor to similarly relaxed architectures.

In this paper, we develop a family of separation logics that can be instantiated on relaxed hardware memory models, and yet 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]. We then instantiate this logic on the Arm-A "user" concurrency model, which 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: r1 := ldr [x]  ||  c: r2 := 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 when actual synchronisation is present — for example, for the Arm-A memory model, along the ordered-before ordering (ob). Different relaxed hardware architectures expose different combinations of ways to impose such synchronisation: address dependencies, release writes, etc. 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 the architecture under consideration 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.

$$\begin{array}{l|l} \multicolumn{2}{c}{x = 0 \land y = 0} \\ a\text{: } r_1 := \mathsf{ldr}\ [x] & c\text{: } r_2 := \mathsf{ldr}\ [y] \\ b\text{: } \mathsf{str}\ [y]\ r_1 & d\text{: } \mathsf{str}\ [x]\ r_2 \end{array}$$

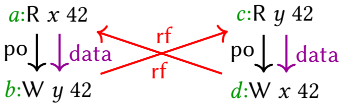


Fig. 2. LB+datas

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, x86, RISC-V, IBM Power, and others, 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 respect for certain syntactic dependencies, ruling out thin-air. These architectural models thus give us a solid foundation that we can reason above.

Second, ultimately, the machine-code binary is what runs — and therefore one 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 in 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 family of separation logics for relaxed hardware architectures, AxSL, that is expressive, supporting reasoning with higher-order ghost state and invariants, and mechanised in Coq, using the Iris program logic framework.

We then instantiate AxSL on two memory models: sequential consistency, and on the Arm-A concurrency model. For both, we use an idealised instruction set architecture (ISA), but our approach is designed to generalise: 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 axiomatic model, so this work offers a path towards similar logics for other architecture axiomatic models (e.g. the RISC-V "user" model, which is similar to that of Arm-A), or, more speculatively, to extensions covering systems semantics, as has been developed for example for Arm [Simner et al. 2022, 2020]. Moreover, both the Arm-A architecture reference manual and RISC-V specify their concurrency architecture in this axiomatic style [Arm Ltd. 2023, Ch.B2] and are actively maintained and occasionally changed, so (while semantics for new features have been developed in multiple styles), it is desirable to be able to track the reference-manual version with minimal effort.

Plan. We describe the program-logic and relaxed-memory context in §2. We explain the key ideas of our logic informally in §3. In §4, we describe the two languages we consider, and how we give their semantics in a way that makes it possible to build an expressive logic featuring higher-order ghost state. We present one language for SC, combining a simplified assembly language with sequential consistency; and one for Arm-A, combining a simplified assembly language featuring dependencies with the real LB-permitting Arm-A axiomatic concurrency model.[1] In §5, we describe the rules of our AxSL logic and exercise them on small, representative examples. We do this in three stages, first we present how to deal with axiomatic memory models ignoring relaxed memory, then we show how to structure the logic to deal with a relaxed memory model but in the simple setting of sequential consistency, and finally we deal with an actual relaxed memory model, namely that of Arm-A. In §6, we define the model of AxSL in Iris, following the same three stages, and present our non-standard definition of weakest precondition. In §7, we present our non-standard proof of adequacy of AxSL in Iris. In §8, we discuss some technical aspects and limitations of our work. We discuss related work in §9, and how our work can be used and extended further in §10. The Arm-A instance of AxSL, its soundness and the examples are formalised in Coq using the Iris separation logic framework; the full development is available at https://github.com/logsem/AxSL.

Difference with the original paper. This article is an extended version of the original paper presented at POPL 2024 [Hammond et al. 2024]. In particular, it makes our contributions more accessible, especially to those who are less familiar with relaxed memory and the memory model of Arm-A, it elaborates the definitions to be self-contained, and it makes technical improvements to the proof technique. In detail:

- We introduce our novel 'opax' type of semantics using a simple language with a simple memory model (namely sequential consistency) in §4.3.3.
- We explain the novel ideas of AxSL$^{\mathsf{Arm}}$ in that simpler setting, building two logics for that simple SC language: AxSL$^{\mathsf{SC}}$ and AxSL$^{\mathsf{SCExt}}$. These two simpler logics work as explanatory steps when building up the syntax and the semantic model of the AxSL$^{\mathsf{Arm}}$.
  - We show how to define a first straightforward logic, AxSL$^{\mathsf{SC}}$, on top of our novel 'opax' style of semantics in §5.2, and how to define a semantic model for it in §6.3.
  - We then show how to define a second, more elaborate logic, AxSL$^{\mathsf{SCExt}}$, that uses the ideas that make AxSL$^{\mathsf{Arm}}$ work in the setting of relaxed memory, but still in the simple setting of sequential consistency in §5.3, and present its semantic model in §6.4.

[1] 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.

- We give a self-contained presentation of the definitions of the model of AxSL$^{\mathsf{Arm}}$, using precise definitions that were omitted in the original paper because of space constraints, in §6.5.
- We expand the explanation of various technical definitions and proofs, and add illustrations to make the technical material more accessible.
- We refine the model of AxSL$^{\mathsf{Arm}}$, leading to some technical improvements that we elaborate on in §8.1, in particular a better proof of adequacy.

Moreover, by demonstrating the ideas of AxSL$^{\mathsf{Arm}}$ on a different (albeit simple) memory model, we have shown that our novel approach to defining semantics and program logics generalises.

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, 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.

We base the current work on axiomatic models. In these, a program gives rise to a large set of candidate complete execution graphs, each with a function from event IDs to events, and a program order relation over event IDs (po) within each thread, and various other base relations. An axiomatic concurrency model typically defines compound relations derived from these, e.g. for Arm-A — which we will use as our main case study — the model of Fig. 4 defines an *observed before* (ob) relation that captures synchronisation, 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,

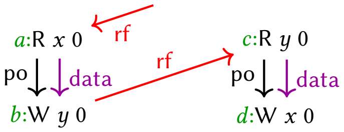


Fig. 3. An SC execution of LB+datas

```
(* 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.

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 (rf) the initial state in Fig. 3 is allowed.

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).

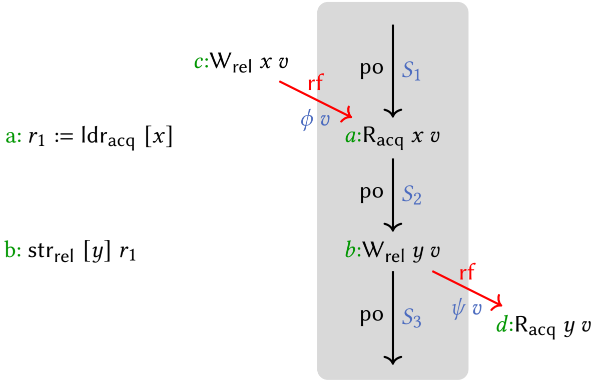


Fig. 5. Thread-local part (in grey) of a candidate execution, annotated with the logical resources in blue flowing on edges. The thread program is on the left. Resources $S_{1,2,3}$ are those in hand at each program point, and the protocol specifies the resources ϕ v and ψ v passed along the release-acquire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

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 §9.

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.

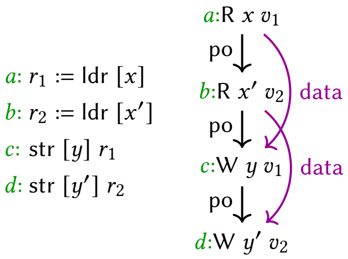


Fig. 6. Intra-thread concurrency

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.

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. Concretely, a load a into register r of a value v, from some location following a protocol ϕ, will give us

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

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 $a \hookrightarrow (\phi\ v)$ 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 §5.6). 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 $a \hookrightarrow P$ 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.

```
1   ${r_1 \overset{\mathrm{r}}{\mapsto} \_ * R_0}$ // $S_1$
2   $a$: $r_1$ := ldr [$x$]
3   ${\exists v_1 .\, r_1 \overset{\mathrm{r}}{\mapsto} v_1@\{a\} * (a \hookrightarrow (\phi\ v_1)) * R_0}$ // $S_2$
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.

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 $a \hookrightarrow (\phi\ v_1)$. When performing the store b on line 4, the proof rule requires us to establish the corresponding flow implication. 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 $a \hookrightarrow P$, we can split $a \hookrightarrow (P * Q)$ into $(a \hookrightarrow P) * (a \hookrightarrow Q)$ and frame the latter off, as in Fig. 8, see §6.5.

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

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[2]. 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,

[2] 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.

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 semantics 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 (§6.5) and a non-standard proof of adequacy (§7) (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.

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$^{\mathsf{Arm}}$ using our new assertions, we then need an adequacy theorem (§7), 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 po ∪ 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 LANGUAGES

As the focus of this paper is on real-world concurrency rather than realistic instruction set architectures, we consider a simplified assembly language, TinyArm, 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 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.

Besides TinyArm, Fig. 9 also depicts the syntax of TinySc, an even simpler language in which we write concurrent programs for a sequentially consistent (SC) memory model. We use this compact language to demonstrate the core idea of opax which is the formal foundation that two logics in §5 build upon. TinySc is syntactically a sublanguage of

$$
\begin{array}{ll}
i_{\mathsf{TinyArm}} ::= & \text{instructions} \\
\quad \mathsf{nop} & \\
\mid r := t & \text{register assignment} \\
\mid \mathsf{br}\ x & \text{branch to address } x \\
\mid \mathsf{bne}\ t\ x & \text{conditional branch} \\
\mid r := \mathsf{ldr}_{os,vr}\ [t_{\mathrm{addr}}] & \text{memory load} \\
\mid r := \mathsf{str}_{os,vr}\ [t_{\mathrm{addr}}]\ t_{\mathrm{data}} & \text{memory store} \\
\mid \mathsf{dmb\ sy} \mid \mathsf{dmb\ st} \mid \mathsf{dmb\ ld} \mid \mathsf{isb} & \text{Arm-A fences}
\end{array}
\qquad
\begin{array}{lcl}
\multicolumn{3}{l}{r \in \mathrm{Reg} \triangleq \{r_0,\ r_1,\ \ldots\}} \\
\multicolumn{3}{l}{v, x \in \mathrm{Word} \triangleq 0..2^{64} - 1} \\
op & ::= & + \mid - \mid \times \\
t & ::= & v \mid r \mid t_1\ op\ t_2 \\
os & ::= & \mathsf{plain} \mid \mathsf{relacq} \mid \mathsf{weakacq} \\
vr & ::= & \mathsf{nexcl} \mid \mathsf{excl}
\end{array}
$$

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

$$
i_{\mathsf{TinySc}} ::= \mathsf{nop} \mid r := t \mid \mathsf{br}\ x \mid \mathsf{bne}\ t\ x \mid r := \mathsf{ldr}\ [t_{\mathrm{addr}}] \mid r := \mathsf{str}\ [t_{\mathrm{addr}}]\ t_{\mathrm{data}}
$$

Fig. 9. Instructions and syntactic sugar of TinyArm and TinySc

TinyArm, where we elide the *os* and *vr* and omit the barriers. Therefore, in the rest of this section, we present their semantics by detailing one and then merely explaining how the other relates to it.

4.1 The Elaboration Semantics of Instructions into the Sail Outcome Interface

The Sail outcome interface defines the intra-instruction semantics for each instruction, independently 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.

We show how to elaborate TinyArm instruction using the interface, especially how to handle Arm's architectural dependencies, and then how to reuse the elaboration for TinySc instructions by merely ignoring those Arm specifics.

4.1.1 The Elaboration of TinyArm *Instructions.* Given an ordering strength *os*, a variety (exclusive or non-exclusive) *vr*, and address x, and dependencies $d \in \mathrm{Dep} \triangleq P(\mathrm{Reg}) \times P(N)$ (composed of register dependencies r, and intra-instruction event dependencies m), MemRead *os vr x d* has type (roughly) Outcome Word, wrapped in the instruction monad IMon A which has constructors $\mathsf{Next}_T : \mathrm{Outcome}\ T \longrightarrow (T \longrightarrow \mathrm{IMon}\ A) \longrightarrow \mathrm{IMon}\ A$ and $\mathrm{Ret} : A \longrightarrow \mathrm{IMon}\ A$. 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, non-exclusive load $r := \mathsf{ldr}\ [t_{\mathrm{addr}}]$, into register r from address t_{addr}, elaborates into a plain (and non-exclusive) memory read from that address with address dependencies given by the auxiliary function $\mathbb{D}[\![-]\!]$ (in our simplified language, dependencies can be computed from the syntax), the value of which is bound to v, followed by a register write to r of v with dependencies $\langle \emptyset, \{0\} \rangle$ meaning that the register write has no register dependency and a dependency on the 0th MemRead of the instruction. See the corresponding reduction rule A-REG-WRITE of our semantics in §4.3 for how the real dependencies are computed from this bookkeeping type Dep. Finally, the elaboration is followed by a program counter increment (which we write with a bind $\gg=$ to be systematic).

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

$$\begin{aligned}
\mathbb{T}\llbracket v \rrbracket &\triangleq \mathsf{Ret}\ v & \mathbb{D}\llbracket v \rrbracket &\triangleq \emptyset \\
\mathbb{T}\llbracket r \rrbracket &\triangleq \mathsf{Next}\ (\mathsf{RegRead}\ r)\ \mathsf{Ret} & \mathbb{D}\llbracket r \rrbracket &\triangleq \{r\} \\
\mathbb{T}\llbracket t_1\ op\ t_2 \rrbracket &\triangleq \mathbb{T}\llbracket t_1 \rrbracket \gg\!= \lambda v_1.\, \mathbb{T}\llbracket t_2 \rrbracket \gg\!= \lambda v_2.\, \mathsf{Ret}\ (v_1\ \mathbb{O}\llbracket op \rrbracket\ v_2) & \mathbb{D}\llbracket t_1\ op\ t_2 \rrbracket &\triangleq \mathbb{D}\llbracket t_1 \rrbracket \cup \mathbb{D}\llbracket t_2 \rrbracket
\end{aligned}$$

Fig. 10. A few cases of the elaboration of TinyArm into the outcome interface (eliding some details), where ldr is the syntactic sugar of $\mathsf{ldr}_{\mathsf{plain,nexcl}}$ and stlr is the syntactic sugar of $\mathsf{str}_{\mathsf{rel,nexcl}}$. The computation of intra-instruction dependencies is highlighted.

A non-exclusive store release stlr $[t_{\mathrm{addr}}]$ t_{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 $\mathbb{T}\llbracket - \rrbracket$, followed by a release (and non-exclusive) memory write to x of v with address data dependencies 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.

4.1.2 The Elaboration of TinySc *Instructions.* We just reuse the elaborations of plain, non-exclusive load and store of TinyArm as the elaborations of load and store of TinySc respectively, and identical elaborations for other shared instructions. The elaboration is further simplified by ignoring register dependencies (defining $\mathbb{D}\llbracket - \rrbracket$ to $\lambda\_.\, \emptyset$), since register dependencies do not matter for SC. For the sake of simplicity, we omit the constant arguments of microinstructions, and for example only write MemRead x for MemRead plain nexcl x $\emptyset$, when we present the microinstructions for TinySc.

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, which ensures that all the pre-executions are consistent with (the instruction semantics of) the program. 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

As discussed in §3.2, it is not clear how to define a syntax-directed program logic over this axiomatic style of semantics. Hence, we reformulate the combination of axiomatic model and instruction semantics in our novel *opax* semantics, mixing *op*erational and *ax*iomatic styles. We first present the language-agnostic shape of this new style of semantics, and then give concrete opax semantics definitions for TinySc and TinyArm. Similarly to how the instruction elaboration of TinyArm extends the instruction elaboration of TinySc with bookkeeping, the instantiation of TinyArm as an opax semantics extends the instantiation of TinySc with bookkeeping local states for register dependencies, etc.

4.3.1 Opax Candidate Executions. Unlike the conventional definition of candidate execution, an *opax candidate execution* has a different meaning. It consists of the usual events and relations, but we swap the validity check with the program consistency check. That is, an opax candidate execution has to be well-formed and satisfies the axiomatic-model validity predicate, but is not assumed to be consistent with a program. We define it using the outcome interface. Formally, an opax candidate execution (graph) X comprises a collection of events in the form of a function lab from event IDs (of type *Eid*) to events, and various relations between events of type $Eid \times Eid$, where an event is of type Outcome $T \times T$, that is, a pair of an outcome request and its response. The validity predicate usually consists of acyclicity requirements on compound relations (defined using base relations), and differs from model to model. The well-formedness condition for relations is standard, except for po, for which the constraint depends on the implementation of *Eid* (we elaborate on this soon). The semantics of a program is then defined as the set of opax candidate executions consistent with the program. We give two formal instantiations following this new notion below.

Opax Candidate executions of SC. An opax candidate execution for SC is defined as

$$X \triangleq \langle \mathsf{lab}, \mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{fr}, \mathsf{sc} \rangle$$

with reads of arbitrary values and writes of values, and with arbitrary reads-from (rf), coherence (co), and fr (equivalent to $\mathsf{rf}^{-1}; \mathsf{co}$) relations between them. The validity requirement is the acyclicity of $\mathsf{sc} \triangleq \mathsf{po} \cup \mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr}$, where $e\ \mathsf{sc}\ e'$ means event e happens before e'.

Opax Candidate executions of Arm-A. An opax candidate execution for Arm-A is defined as

$$X \triangleq \langle \mathsf{lab}, \mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{fr}, \mathsf{ctrl}, \mathsf{addr}, \mathsf{data}, \mathsf{rmw}, \ldots (\text{compound relations}) \rangle$$

with new ctrl, data, and addr dependencies, and rmw base relation for exclusives. The compound relations and validity requirements are defined in Fig. 4.

4.3.2 The Shape of the Opax Semantics. The opax semantics works in two phases. In the first phase, execution of a whole program starts by guessing a complete opax candidate execution graph X for the program. This execution graph is well-formed and valid, but is otherwise unconstrained, and in particular is for now unrelated to the program itself.

In the second phase, each thread is executed independently: interaction happens 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 $c_1, \ldots, c_n$. (Instruction fetching could be accurately modelled in the memory model, as per Simner et al. [Simner et al. 2020], but this would lead to significant complexity.)

Each thread state s is either Ctd C ("continued"), which represents an ongoing thread execution, or Done T, which represents a completed thread execution. Here C is a tuple $\langle p, T \rangle$ where p is the remaining microinstruction program in the Sail outcome interface for the current instruction, and T is the thread state which is dependent on the language and the concurrency model.

For each thread tid, $s_{\mathrm{init}}(c_{tid})$ is its initial thread state, with pc set to c_{tid} 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, and are deterministic.

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

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

It is worth-noting that the only source of non-determinism is the guessing step; all following thread-local reductions are deterministic. A similar pattern appears in operational semantics with a quantified scheduler, where picking the scheduler is non-deterministic, and the interleaving is determined by the scheduler.

A stuck thread is not an error state: it rather indicates that the guessed graph does not correspond to this program. Rule Whole-system-execution of our semantics ignores these wrongly guessed graphs, leaving only the execution graphs of the program.

In the rest of this subsection, we give the instantiations of the opax semantics for the two languages. We start with the semantics of TinySc, which is compact and makes it easy to demonstrate the core idea, and then show the semantics of TinyArm, explaining how to handle the complexities that come with Arm-A: access kinds and dependencies.

4.3.3 *Opax Semantics for* TinySc.

For TinySc, we instantiate thread state T with a tuple $\langle regs, IT \rangle$ to track a register state *regs* - a finite map from register names to values - and an intra-instruction state IT which comprises a thread local event counter *cntr*.

We sketch selected rules of the thread operational semantics in Fig. 11. A thread executes by executing the current microinstruction program until it ends, at which point all microinstructions of the current instruction must have been executed (checked by instr-done), and then (rule S-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. The program execution terminates when pc is outside the instruction memory (rule S-term), at which point there must not be further events by this thread in the graph (checked by prog-done).

The event identifier of the current microinstruction $e = \langle tid, IT.cntr \rangle$ comprises a thread identifier (zero being reserved for the 'initial' thread that contains all the initial writes), and the event counter $IT.cntr$ which is an ordered pair comprising an instruction counter and an intra-instruction event counter.

A MemRead microinstruction can execute (rule S-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 have the appropriate po edges to it. To check po edges, the intra-instruction counter of $IT.cntr$ gets incremented with next-e after executing every microinstruction; the instruction counter gets incremented with next-i, which additionally resets the intra-instruction counter when finishing up an instruction (in S-reload). po edges are special, in the sense that a po edge between two non-initial events can be checked for by determining whether their identifiers have same thread id and their local event counter values are lexicographically ordered. This is part of the well-formedness condition of the execution graph. This indicates that we do not need to explicitly check if there is a po edge between two events in the graph – we only need to increment the counters correctly.

A RegWrite microinstruction can similarly execute (rule S-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 *regs* on the value of the write. (This register event is not used in the axiomatic memory models for either SC or Arm-A, because the former does not need it and the latter instead uses primitive dependency relations, but the interface includes it to support operational models and other axiomatic models.)

$$\textsc{S-mem-read}\quad \frac{X.\mathsf{lab}(e) = \mathsf{R}\ x\ v \qquad e = \langle tid, IT.cntr\rangle}{\mathsf{Ctd}\ \langle \mathsf{Next}\ (\mathsf{MemRead}\ x)\ K, \langle regs, IT\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle K\ v, \langle regs, \mathsf{next\text{-}e}(IT)\rangle\rangle}$$

$$\textsc{S-mem-write}\quad \frac{X.\mathsf{lab}(e) = \mathsf{W}\ x\ v \qquad e = \langle tid, IT.cntr\rangle}{\mathsf{Ctd}\ \langle \mathsf{Next}\ (\mathsf{MemWrite}\ x\ v)\ K, \langle regs, IT\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle K\ (), \langle regs, \mathsf{next\text{-}e}(IT)\rangle\rangle}$$

$$\textsc{S-reg-write}\quad \frac{X.\mathsf{lab}(e) = \mathsf{RegW}\ r\ v \qquad e = \langle tid, IT.cntr\rangle}{\mathsf{Ctd}\ \langle \mathsf{Next}\ (\mathsf{RegWrite}\ r\ v)\ K, \langle regs, IT\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle K\ (), \langle regs[r \mapsto v], \mathsf{next\text{-}e}(IT)\rangle\rangle}$$

$$\textsc{S-reg-read}\quad \frac{X.\mathsf{lab}(e) = \mathsf{RegR}\ r\ v \qquad e = \langle tid, IT.cntr\rangle \qquad regs(r) = v}{\mathsf{Ctd}\ \langle \mathsf{Next}\ (\mathsf{RegRead}\ r)\ K, \langle regs, IT\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle K\ (), \langle regs, \mathsf{next\text{-}e}(IT)\rangle\rangle}$$

$$\textsc{S-reload}\quad \frac{regs(\mathsf{pc}) = x \qquad I(x) = opcode \qquad \mathsf{decode}(opcode) = p \qquad \mathsf{instr\text{-}done}(X, IT.cntr)}{\mathsf{Ctd}\ \langle \mathsf{Ret}\ (), \langle regs, ?R, IT\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle p, \langle regs, \mathsf{next\text{-}i}(IT)\rangle\rangle}$$

$$\textsc{S-term}\quad \frac{regs(\mathsf{pc}) = x \qquad x \notin \mathsf{dom}(I) \qquad \mathsf{prog\text{-}done}(X, IT.cntr)}{\mathsf{Ctd}\ \langle \mathsf{Ret}\ (), \langle regs, IT\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Done}\ \langle regs, IT\rangle}$$

Fig. 11. Selected reduction rules of opax semantics for TinySc

4.3.4 Opax Semantics for TinyArm. We define the opax semantics for TinyArm by extending the TinySc semantics with instrumentation to track Arm-A dependencies and access kinds. Concretely, we first extend the thread state T with the set $srcs_{\mathsf{ctrl}}$ of sources of control dependencies, and the previous exclusive read event e_{rmw}. Then, we augment every register in *regs* with dependency information alongside its value. Finally, we add the list of identifiers of intra-instruction read events seen so far *mrd* as a new field of IT.

The two selected rules in Fig. 12 illustrate how these extensions check dependency edges. A non-exclusive MemRead microinstruction now has to have the appropriate addr and ctrl edges to it (rule A-mem-read-nexcl), as checked using the thread state (the set of data dependencies d_{addr} and the control dependency sources $srcs_{\mathsf{ctrl}}$ respectively). Unlike how we treat po, here we have to check if those dependency edges exist in the graph explicitly. The event ID e is appended to the intra-instruction memory read list $IT.mrd$ by auxiliarly function intra-read-app, so that later register microinstructions can obtain e by providing a position in the list to check a dependency from e if the register value is computed from the read value v (we demonstrate this with the elaboration of load in Fig. 10 in the following paragraph).

A RegWrite microinstruction (rule A-reg-write) now also updates the dependency of the register d_{reg} for the local registers in *regs*. The dependency d_{reg} is a set of event identifiers computed from two sources tracked with d: the union of the dependencies of every register in $d.\mathsf{r}$ (the left iterated union), and intra-instruction event dependencies that are memory reads whose indices are in $d.\mathsf{m}$ (the right iterated union). For instance, in the elaboration of load in Fig. 10 where d is instantiated to $\langle \emptyset, \{0\}\rangle$, d_{reg} is computed to be $\{e\}$ when $IT.mrd$ is $[e]$ (that is, we take the 0th event from the list), where e is the event ID of the memory read event preceding the register write. Therefore, we conclude that the data of the register v comes from e, and update *regs* accordingly.

4.3.5 Stuckness. 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 discard stuck executions by assuming unstuckness in the definition

A-MEM-READ-NEXCL

$$\frac{\begin{array}{c} X.\mathsf{lab}(e) = \mathsf{R}_{os,\mathsf{nexcl}}\ x\ v \qquad e = \langle tid, IT.cntr\rangle \qquad \{\langle e_d, e\rangle \mid e_d \in srcs_{\mathsf{ctrl}}\} = to(e, X.\mathsf{ctrl}) \\ \{\langle e_d, e\rangle \mid r \in d_{\mathsf{addr}}.\mathsf{r} \land regs(r) = \langle \_, srcs_d\rangle \land e_d \in srcs_d\} = to(e, X.\mathsf{addr}) \qquad \mathsf{intra\text{-}read\text{-}app}(IT, e) = IT' \end{array}}{\mathsf{Ctd}\ \langle \mathsf{Next}\ (\mathsf{MemRead}\ os\ \mathsf{nexcl}\ x\ d_{\mathsf{addr}})\ K, \langle regs, srcs_{\mathsf{ctrl}}, ?R, IT\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle K\ v, \langle regs, srcs_{\mathsf{ctrl}}, ?R, \mathsf{next\text{-}e}(IT')\rangle\rangle}$$

A-REG-WRITE

$$\frac{X.\mathsf{lab}(e) = \mathsf{RegW}\ r\ v \qquad e = \langle tid, IT.cntr\rangle \qquad d_{\mathsf{reg}} = \left(\bigcup_{\{srcs_d \mid r \in d.\mathsf{r} \land regs(r) = \langle \_, srcs_d\rangle\}} srcs_d\right) \cup \left(\bigcup_{i \in d.\mathsf{m}} \{IT.mrd[i]\}\right)}{\mathsf{Ctd}\ \langle \mathsf{Next}\ (\mathsf{RegWrite}\ r\ v\ d)\ K, \langle regs, ?R, IT\rangle\rangle \xrightarrow{tid,X,I}_{\mathsf{h}} \mathsf{Ctd}\ \langle K\ (), \langle regs[r \mapsto \langle v, d_{\mathsf{reg}}\rangle], ?R, \mathsf{next\text{-}e}(IT)\rangle\rangle}$$

Fig. 12. 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 target e. The instrumentation to deal with Arm dependencies is highlighted in orange.

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.

4.3.6 *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 LOGICS

The goal of this section is to build up to AxSL$^{\mathsf{Arm}}$. We do this incrementally, introducing two intermediate logics to explain the different building blocks of AxSL$^{\mathsf{Arm}}$.

(1) We start by tackling only the challenge of defining a logic on top of an opax semantics, and illustrate it with our first logic: AxSL$^{\mathsf{SC}}$. We start from a simple setting: sequentially consistent (SC) concurrency: the question here is how to deal, in a logic like Iris, with a fixed execution graph representing the shared memory. This is merely the first step: AxSL$^{\mathsf{SC}}$ is built on the right foundations (namely, an opax semantics) to scale to relaxed concurrency, but it still bakes in too much ordering in its structure.

(2) We then move on to describe our novel style of assertions compatible with relaxed memory, and illustrate it with our second logic: AxSL$^{\mathsf{SCExt}}$, a logic with both foundations and an assertion style compatible with relaxed concurrency. For simplicity, AxSL$^{\mathsf{SCExt}}$ stays in the context of sequential consistency, but follows the fine-grained resource management style sketched in §2. In particular, AxSL$^{\mathsf{SCExt}}$ employs 'tied-to' assertions and flow implications in order to be compatible with relaxed concurrency.

(3) Finally, we present AxSL$^{\mathsf{Arm}}$, our logic for the relaxed memory of Arm. The final challenge is to deal with the subtleties of such a memory model: syntactic dependencies, external vs. internal reads, exclusives, etc.

For each of the three logics, we present a selection of its proof rules followed by examples. The proof rules (and the Hoare triples used in them) of the three logics are defined at two abstraction levels: the underlying rules are for microinstructions, and are proven sound against the semantics of Hoare triples described in §6.5, while the high-level rules explicitly used in proofs of programs are for the surface instructions of Fig. 9, derived from the former by reasoning about instruction elaboration (§4.1). Before diving into the three logics, we first discuss the resource transfer mechanism that they employ.

5.1 Resource Transfer with Protocols

Concurrent separation logic (CSL) uses invariants to share and transfer resources. However, invariants are unsound in the context of relaxed concurrency [Dang et al. 2020; Turon et al. 2014]. Intuitively, this is because, with relaxed concurrency, threads may have different views on the shared memory, thus owning potentially inconsistent resources describing those views, while classic CSL invariants require the transferred resources to be consistent across all threads.

Furthermore, even though invariants have been shown to work well with heap reasoning, it is unknown how they work in conjunction with graph reasoning. Recall that in CSL, to share memory resources (e.g. some points-tos), one usually allocates some invariant with them, and then distributes the duplicates of the invariant to the threads. The threads then may obtain the ownership of the resources (temporarily) by opening the invariant for loading and storing. In the graph-based approach that we present in this section, one is not required to own any shared memory assertions to access the shared memory; instead, one makes *assertions* about the graph representing memory. It is not clear to us how this can be made to work with invariants, in particular how to make connections between the newly-gained graph assertions and the resources shared by invariants.

In AxSL, we instead, inspired by previous relaxed memory logics including RSL and GPS, use a notion of protocol for resource transfer (compatible with the invariants we use for exclusives in §5.7). A protocol Φ is a rely/guarantee-style protocol that enables thread-local reasoning by expressing the intended resource transfer across an entire program. For each location x, $\Phi(x)$, referred to as a per-location protocol, is a simple variant of the per-location invariants of GPS. (For simplicity, we only consider static protocols with a single state) We discuss their relation in detail in §9. Our Hoare triples are simply parametrised by this fixed protocol, which should be agreed upon by all threads.

Our triples ensure the protocol $\Phi(x)$ holds at every event involving location x, which enables resource transfer between those events. In particular, as we will see in the proof rules, every read event of x obtains the resource specified by $\Phi(x)$ from the external write that it is reading from, and every write must supply that resource. A protocol for x takes as arguments a value v and an event ID e. The event ID e is associated with the write event that fullfills the protocol. This event ID argument e is used for explicit reasoning about the execution graph, as illustrated in the message passing example below. 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.

5.2 Dealing with Opax Semantics: The AxSL$^{\text{SC}}$ Logic

AxSL$^{\text{SC}}$ is a thin logical layer above our opax semantics of TinySc. It essentially exposes all the details of the opax semantics to the users of the logic, which gives the logic an unique shape and new reasoning principles compared to classic operational-based CSLs. The distinction primarily comes from the diverging representations of the shared memory of the underlying semantics. Since the opax semantics guesses a fixed execution graph upfront, in AxSL$^{\text{SC}}$, we can only deal with persistent knowledge on graph events and memory orders, rather than the usual points-to assertions that represent the state of a shared and dynamic heap at individual locations (We show how to close this gap by recovering points-tos in AxSL$^{\text{SC}}$ in Appendix B). To understand how AxSL$^{\text{SC}}$ works, in particular its graph-based compositional reasoning, we show some of its proof rules, and then verify a message passing example.

SC-HT-MICRO-MEMREAD

$$\left\{❶\mathsf{PoPred}(e_{\mathrm{po}}) * \forall e, v, e_w.\left(\begin{array}{l}❷\mathsf{GraphFactsR}(e, x, v, e_w, e_{\mathrm{po}}) * ❸\Phi(x, v, e_w)\\ \quad\Rrightarrow ❹P(e, x, v, e_w, e_{\mathrm{po}}) * \Phi(x, v, e_w)\end{array}\right)\right\}$$

$$\mathsf{MemRead}\ x$$

$$\{v.\, \exists e, e_w.\, ❺\mathsf{PoPred}(e) * \mathsf{GraphFactsR}(e, x, v, e_w, e_{\mathrm{po}}) * P(e, x, v, e_w, e_{\mathrm{po}})\}_{tid,\Phi}$$

SC-HT-MICRO-MEMWRITE

$$\{❶\mathsf{PoPred}(e_{\mathrm{po}}) * \forall e.\, ❷\mathsf{GraphFactsW}(e, x, v, e_{\mathrm{po}}) \Rrightarrow ❸\Phi(x, v, e)\}$$

$$\mathsf{MemWrite}\ x\ v$$

$$\{().\, \exists e.\, ❹\mathsf{PoPred}(e) * \mathsf{GraphFactsW}(e, x, v, e_{\mathrm{po}})\}_{tid,\Phi}$$

SC-HT-MICRO-REGWRITE

$$\{r \stackrel{\mathrm{r}}{\mapsto} \_\}\mathsf{RegWrite}\ r\ v\ \{().\, r \stackrel{\mathrm{r}}{\mapsto} v\}_{tid,\Phi}$$

Fig. 13. Seleted proof rules of AxSL$^{\mathrm{SC}}$

5.2.1 *Proof Rules for Microinstructions.*

Fig. 13 depicts three slightly specialised proof rules for microinstructions MemRead, MemWrite, and RegWrite. We first explain the rule SC-HT-MICRO-MEMREAD for MemRead in detail, which is proved sound against the opax rule S-MEM-READ.

Our microinstruction Hoare triple for MemRead has the form $\{P\}\mathsf{MemRead}\ x\ \{v.\, Q\}_{tid,\Phi}$, 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 passed to the continuation to continue reasoning about the thread. The event ID of the associated memory read event is existentially quantified in Q as e.

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, we use a bookkeeping assertion ❶ $\mathsf{PoPred}(e_{\mathrm{po}})$ to capture the e_{po} parts of the opax thread state T. Intuitively, the assertion keeps track of the event that will become the po immediate predecessor of the memory event e that associates with the next microinstruction, and thus allow us to conclude facts about new incoming po edge e_{po} po e, that is included in $\mathsf{GraphFactsR}(e, x, v, e_w, e_{\mathrm{po}})$. It gets updated accordingly in the postcondition, as ❺. The predicate ❷ $\mathsf{GraphFactsR}(e, x, v, e_w, e_{\mathrm{po}})$ includes all graph assertions we can conclude for the read event. Besides the incoming po edge, it also includes an event assertion e:R x v indicating that the event is assigned with ID e, and another edge assertion e_w rf e to relate the read with the write e_w that it is reading from. It is worth noting that all graph assertions are persistent knowledge that can be freely duplicated, which reflects the fact that the graph is fixed once guessed in opax. Furthermore, we can stop the reasoning (by contradiction) if some graph facts gathered together imply a violation of the validity predicate of the axiomatic memory model, since this further implies that we are reasoning about a graph that does not represent a valid execution result of the program. We demonstrate this idiom in the message passing example below.

Second, to support high-level reasoning via resource transfer, the precondition has a user-supplied ❹ $P(e, x, v, e_w, e_{\mathrm{po}})$ on the right side of an Iris view shift $\Rrightarrow$ (a separation implication that permits resource update), which also appears in the post condition, meaning that P is obtained after the read. This view shift constrains what we can conclude in ❹ given the facts about the new read and the protocol resource ❸ received from the write e_w being read. Note that we have to give back the same protocol resource Φ after concluding P, to ensure that the resource remains available for other potential reads of the same write e_w. In a sense, this view shift allows us to *temporarily* obtain the ownership of

concrete addresses	symbolic addresses	instruction instances	memory events
1000:	a:	str $[x]$ 42	a:W x 42
1000 + 4:	b:	$\mathsf{str}_{\mathsf{rel}}$ $[y]$ 1	b:$\mathsf{W}_{\mathsf{rel}}$ y 1
1000 + 8:	c:	r := ldr $[z]$	c:R z 2

Fig. 14. Conflating (left columns) a numerical instruction address (1000, 1000+4, 1000+8) with a symbolic instances address (a, b, c), and (right columns) an instruction instance (str $[x]$ 42, ...) with its unique memory event (W x 42, ...).

the protocol resource of e_w at this memory read, akin to CSL invariants that can only be opened for a single program step. As we will see later in $\text{AxSL}^{\text{SCExt}}$, this view shift is a specialised form of the notion of flow implication.

The rule SC-HT-MICRO-MEMWRITE for MemWrite is similar, except that Φ is on the right side of the view shift, indicating that it is sent away.

The rule SC-HT-MICRO-REGWRITE updates a points-to assertion for register r to mirror the change to $T.regs$ in the opax rule A-REG-WRITE.

5.2.2 Proof Rules for Instructions. Moving from microinstructions to the instructions composed out of them, we can write a derived high-level proof rule for each instruction. These high-level rules can be further specialised to specific programming idioms and their assumptions to make reasoning practical.

Before looking at the proof rules, 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 $\{P\}\ a\colon i\ \{a'\colon Q\}_{tid,\Phi}$, where a is the address of the instruction, and a' is the address of the next instruction. This instruction triple is implemented using register points-to of pc: the precondition P is combined with $\text{pc} \overset{\text{r}}{\mapsto} a$, and Q is combined with $\text{pc} \overset{\text{r}}{\mapsto} a'$ for the appropriate a' — which is $a + 4$ (as per the elaboration of IncPC) except for branch instructions.

For presentation purposes, for programs without branching (and thus no looping), we can (as illustrated in Figure 14) conflate instruction instances with instructions, and thus conflate instruction identifiers a, b, c, etc. with numerical addresses for instructions in memory $a, a + 4, a + 8$, etc. For languages 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 opax 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).

For instance, the rule SC-HT-INST-LDR in Fig. 15 is for a load instruction at address a with an immediate address x, which is elaborated into MemRead x followed by RegWrite followed by IncPC.

This rule is slightly specialised to only taking an immediate address, and is derived by a SC-HT-MICRO-MEMREAD followed by and SC-HT-MICRO-REGWRITE. Similarly, one can prove a specialised instruction rule SC-HT-INST-STR for a store

SC-ht-inst-ldr

$$\left\{ r \overset{\mathsf{r}}{\mapsto} \_ * \mathsf{PoPred}(e_{\mathsf{po}}) * \forall e, v, e_w.\ \begin{pmatrix}\mathsf{GraphFactsR}(e, x, v, e_w, e_{\mathsf{po}}) * \Phi(x, v, e_w) \\ \Rrightarrow P(e, x, v, e_w, e_{\mathsf{po}}) * \Phi(x, v, e_w)\end{pmatrix} \right\}$$

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

$$\left\{ a + 4\colon r \overset{\mathsf{r}}{\mapsto} v * \exists e_w.\ \mathsf{PoPred}(a) * \mathsf{GraphFactsR}(a, x, v, e_w, e_{\mathsf{po}}) * P(a, v, e_w) \right\}_{tid,\Phi}$$

SC-ht-inst-str

$$\left\{ \mathsf{PoPred}(e_{\mathsf{po}}) * \forall e.\ \left(\mathsf{GraphFactsW}(e, x, v, e_{\mathsf{po}}) \Rrightarrow \Phi(x, v, e)\right) \right\}$$

$$a\colon \mathsf{str}\ [x]\ v$$

$$\left\{ a + 4\colon \mathsf{PoPred}(a) * \mathsf{GraphFactsW}(a, x, v, e_{\mathsf{po}}) \right\}_{tid,\Phi}$$

Fig. 15. Two instruction proof rules with instruction triples. $a + 4$ in the postcondition indicates the address of the next instruciton.

with immediate operators by SC-ht-micro-memwrite. We use these two instruction rules in the message passing example below.

5.2.3 *Message Passing.* We now demonstrate the graph reasoning capability that opax-based logics have, and how we achieve local reasoning with protocols, by exercising AxSL$^{\mathrm{SC}}$ on a message-passing litmus test.

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 receiving thread reads from the flag address to check whether a message has been passed to it, and then reads from the data location.

$$\begin{array}{l||l} a\colon \mathsf{str}\ [\mathit{data}]\ 42 & c\colon r_1 := \mathsf{ldr}\ [\mathit{flag}] \\ b\colon \mathsf{str}\ [\mathit{flag}]\ 1 & d\colon r_2 := \mathsf{ldr}\ [\mathit{data}] \end{array}$$

Fig. 16. MP in SC: $r_1 = 1 \longrightarrow r_2 = 42$

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, $r_1 \overset{\mathsf{r}}{\mapsto} v * r_2 \overset{\mathsf{r}}{\mapsto} v' * (v = 1 \longrightarrow v' = 42)$.

Picking the protocol. The proof sketch of Fig. 17 relies on extensive graph reasoning. We first specify the protocol used in the proof. For the data address, we pick $\Phi(\mathit{data}, v, e) \triangleq \mathsf{Initial}(e) \vee v = 42$, where $\mathsf{Initial}(e)$ 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

$$\Phi(\mathit{flag}, v, e) \triangleq \mathsf{Initial}(e) \vee \left(v = 1 * e{:}\mathsf{W}\ \mathit{flag}\ 1 * \exists e'.\ e'{:}\mathsf{W}\ \mathit{data}\ 42 * e'\ \mathsf{po}\ e\right)$$

requiring that a non-initial write to the flag address is only allowed if it is a write of value 1, on the sending thread, which is po after a write of 42 to the data address.

Note that the only information we pass between threads with this protocol are persistent graph facts. It means that we can always duplicate and keep the complete protocol resources in the thread when reading. With the protocol in hand, we can give a proof sketch for each thread.

Sending 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 has value 42. We apply SC-ht-inst-str twice for the two writes. At the write of the data, we learn $\mathsf{PoPred}(a)$ and $a{:}\mathsf{W}\ \mathit{data}\ 42$ in

```
Sending thread:
1   $\{\mathsf{PoPred}(-)\}$
2       $\mathsf{GraphFactsW}(a, \mathit{data}, 42, -)$
        $\Rrightarrow$
3       $\Phi(\mathit{data}, 42, a) * a{:}\mathsf{W}\ \mathit{data}\ 42$
4 $a$: str [$\mathit{data}$] 42
5   $\{\mathsf{PoPred}(a) * a{:}\mathsf{W}\ \mathit{data}\ 42\}$
6       $\mathsf{GraphFactsW}(b, \mathit{flag}, 1, a) * a{:}\mathsf{W}\ \mathit{data}\ 42$
        $\Rrightarrow$
7       $\Phi(\mathit{flag}, 1, b)$
8 $b$: str [$\mathit{flag}$] 1
9   $\{\mathsf{PoPred}(b) * a{:}\mathsf{W}\ \mathit{data}\ 42 * b{:}\mathsf{W}\ \mathit{flag}\ 1 * a\ \mathsf{po}\ b\}$

Receiving thread:
10  $\{\mathsf{PoPred}(-) * r_1 \overset{\mathsf{r}}{\mapsto} \_ * r_2 \overset{\mathsf{r}}{\mapsto} \_\}$
11      $\forall v, e_w.\ \mathsf{GraphFactsR}(c, \mathit{flag}, v, e_w, -) * \Phi(\mathit{flag}, v, e_w)$
        $\Rrightarrow$
12      $\Phi(\mathit{flag}, v, e_w) * \Phi(\mathit{flag}, v, e_w) * c{:}\mathsf{R}\ \mathit{flag}\ v * e_w\ \mathsf{rf}\ c$
13 $c$: $r_1$ := ldr [$\mathit{flag}$]
14  $\{\mathsf{PoPred}(c) * \exists v, e.\ r_1 \overset{\mathsf{r}}{\mapsto} v * \Phi(\mathit{flag}, v, e) * c{:}\mathsf{R}\ \mathit{flag}\ v * e\ \mathsf{rf}\ c\}$
15      $\forall v', e_w.\ \mathsf{GraphFactsR}(d, \mathit{data}, v', e_w, c) * \Phi(\mathit{data}, v', e_w)$
        $\Rrightarrow$
16      $\Phi(\mathit{data}, v', e_w) * (v = 1 \longrightarrow v' = 42)$
17 $d$: $r_2$ := ldr [$\mathit{data}$]
18  $\{\mathsf{PoPred}(d) * \exists v, v'.\ r_1 \overset{\mathsf{r}}{\mapsto} v * r_2 \overset{\mathsf{r}}{\mapsto} v' * (v = 1 \longrightarrow v' = 42)\}$
```

Fig. 17. Proof sketch of MP in AxSL$^{\mathsf{SC}}$.

line 5. 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. We illustrate how to unfold the two instructions to resources, in particular graph facts, in the proof in Fig. 18.

Receiving thread. The receiving thread is where interesting graph reasoning happens. We apply SC-HT-INST-LDR twice for the two reads. We read from the flag, learning $r_1 \overset{\mathsf{r}}{\mapsto} v$ and $\Phi(\mathit{flag}, v, e)$ for some write e in line 14 (we get Φ by duplicating it in line 12). Finally, we consider the data read. We learn $r_2 \overset{\mathsf{r}}{\mapsto} v'$ and $\Phi(\mathit{data}, v', e_w)$ for some v' and e_w, and are required to prove $v = 1 \longrightarrow v' = 42$. The graph reasoning in line 15 & 16 starts by case splitting on $\Phi(\mathit{data}, v', e_w)$. We have $v' = 42$ immediately in the right case. In the left case, we derive a contradiction from $\mathsf{Initial}(e_w)$ and $v = 1$ with graph facts. That is, an fr from d to e', the write to data in the sending thread, can be derived given e_w being the initial write, which closes a *hb* cycle

$$d\ \mathsf{fr}\ e'\ \mathsf{po}\ e\ \mathsf{rf}\ c\ \mathsf{po}\ d$$

violating the SC axiom.

5.3 Tracking Flow of Resources: The AxSL$^{\mathsf{SCExt}}$ Logic

AxSL$^{\mathsf{SCExt}}$ extends the syntax of AxSL$^{\mathsf{SC}}$ with a notion of tied-to assertion: $a \hookrightarrow P$ means that P is *tied* to event a. This extension allows us to track sources of resources and how resources *flow* between individual events precisely, instead

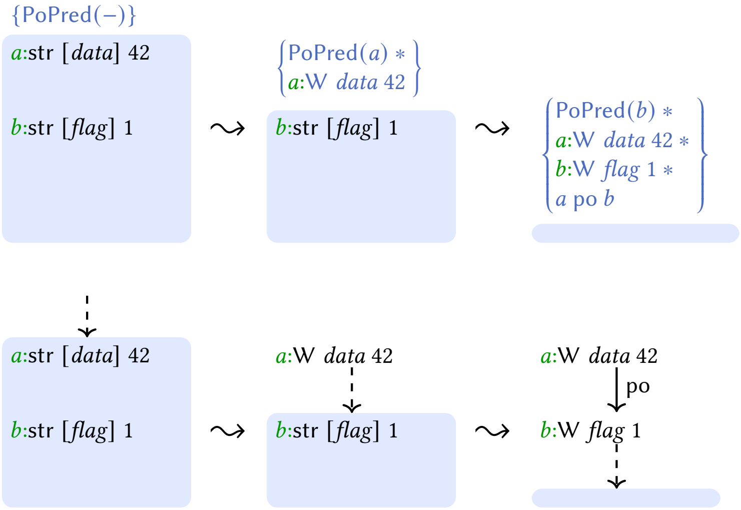


Fig. 18. The proof (here, of the writer side of MP) in progress (proof steps are indicated by $\rightsquigarrow$) unfolds the program (on a blue background) into a graph: either implicitly via assertions $\{\dots\}$ in the top row, or visualised as an explicit graph in the bottom row.

of mixing resources regardless of their sources as in AxSL^{SC}. It is worth noting that this does not add any additional expressiveness power to the logic - in fact AxSL^{SC} is already a fine logic for SC - but is a robust solution that is also applicable to the relaxed concurrency of Arm-A for which an AxSL^{SC}-like construction is unsound. Recall that in SC, sc is the acyclic synchronisation relation, thus passing resources along it (or its subrelations) is sound. In fact, for thread local reasoning, it suffices to flow resources along po, which is included in sc and is also the reasoning order. In AxSL^{SC} (and other CSLs for SC), we implicitly unify the resource flowing order and the reasoning order, but in $\text{AxSL}^{\text{SCExt}}$ we separate them by making the former *explicit* with the tied-to assertion. We elaborate this idea with the $\text{AxSL}^{\text{SCExt}}$ version of AxSL^{SC} rules depicted in Fig. 13.

5.3.1 *Proof Rules for Microinstructions.* There are substantial similarities between $\text{AxSL}^{\text{SCExt}}$ rules and their AxSL^{SC} counterparts. We have the same bookkeeping assertions for the immediate po predecessor, which is updated in the post condition; and the same clauses universally quantified by the new event e accompanied by the graph fact about it. The two main distinctions are the use of the tied-to assertions, and the new FlowSCX predicates for high-level resource transfer between nodes. Let us take a closer look at them in SCExt-ht-micro-memread.

This rule allows us to explicitly reason about resources flowing to e along po edges. If there is such an incoming edge, say from e' to e, and we have an $e' \looparrowright P$, then the flow implication FlowSCR 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 local resources (which need not be persistent) that flow along po edges, as collected in the partial event-to-resource map m (for thread-internal resource flow), combined with the (usually persistent) resources flowing from external events (here, the quantified external write e_w that e is reading from), as specified by the protocol Φ.

To apply the rule, the user has to supply a finite map m to specify how to flow thread-local resources to the current node to show the flow implication (❸). Assertion ❶ $\ast_{(e'_{\text{po}} \mapsto P_{\text{po}}) \in m} (e'_{\text{po}} \looparrowright P_{\text{po}})$, collecting the resources in m for the premise of the flow implication. The map m is constrained by assertion ❷ in the hypothetical reasoning, which requires that an event e'_{po} can only occur in the domain of m when there will be (given the graph facts) an po edge to the

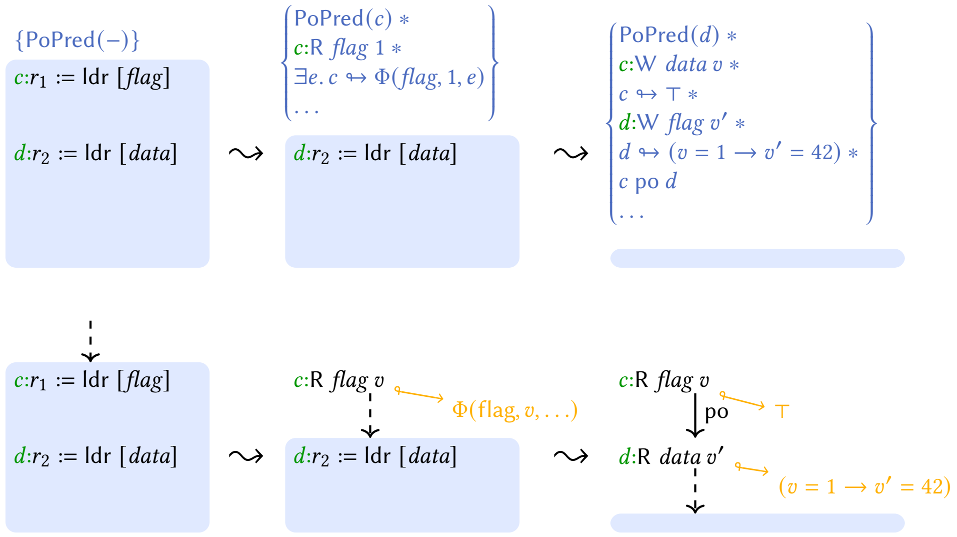


Fig. 19. The proof (here, of the reader side of MP, in AxSL$^{\mathsf{SCExt}}$) in progress. Resources tied-to a node (in orange) 'dangle' off it. Between the second $\rightsquigarrow$, we flow $\Phi(\mathit{flag}, v, \ldots)$ from c to d along po, by consuming it from c and using it to conclude $(v = 1 \longrightarrow v' = 42)$ at d.

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 ③, tied to the new memory event e, as ④. This flow implication FlowSCR replicates the view shift of SC-HT-MICRO-MEMREAD except for the now explicit local resource transfer from po predecessors (the iteratedd separation conjunction). Both this and FlowSCW for MemWrite are instances of the general definition of the flow implication.

5.3.2 Message Passing. We revisit MP to showcase the resource flow reasoning with tied-tos in AxSL$^{\mathsf{SCExt}}$. This time, we use the same protocol, but adapt the specification (changes highlighted in orange) to account for explicit resource flowing, as follows:

$$r_1 \overset{\mathrm{r}}{\mapsto} v * r_2 \overset{\mathrm{r}}{\mapsto} v' * \exists e.\, e \looparrowright (v = 1 \longrightarrow v' = 42) * (\exists e'.\, e \text{ po } e' \lor e = e' * \mathsf{PoPred}(e'))$$

In general, P holds whenever $a \looparrowright P$ appears in a post condition, since one can flow P from a to a hypothetical terminating event along po. In this new specification, the implication $v = 1 \longrightarrow v' = 42$ is embedded in a tied-to for an event e, which is po-before the terminating event, as stated in the last clause.

We look at a proof sketch of the receiving thread depicted in Fig. 21 (with an illustration in Fig. 19); the proof of the sending thread is nearly identical to that of AxSL$^{\mathsf{SC}}$ as no local resource flowing is happening in the thread. This time in the receiving thread, to get the protocol resource $\Phi(\mathit{flag}, \ldots)$ from the *flag* load, we have to justify FlowSCR with argument P being instantiated with Φ, as in line 2 & 3. After that, in line 4, the transferred Φ is tied to the memory read c, which claims a constraint that one can only access Φ if is po after c. To conclude the specification with the graph facts in Φ, we have to flow it to the second load. We therefore let the user-assigned m of the read rule be a singleton map $[c \mapsto \Phi(\mathit{flag}, \ldots)]$, and show that c is indeed a po predecessor of d by $\mathsf{PoPred}(c)$[3]. Then, we show the flow implication FlowSCR for the load of *data* in line 6 & 7, with P being $v = 1 \longrightarrow v' = 42$, where we perform the same

[3] In this example, the protocol resource we flow is persistent, but we can also flow non-persistent resources

SCExt-ht-micro-memread

$$\left\{\begin{array}{l}\mathsf{PoPred}(e_{\mathrm{po}}) * \text{❶}\mathop{\Large\ast}_{(e'_{\mathrm{po}}\mapsto P_{\mathrm{po}})\in m}(e'_{\mathrm{po}} \looparrowright P_{\mathrm{po}}) * \\ \forall e, v, e_w.\left(\begin{array}{l}\mathsf{GraphFactsR}(e, x, v, e_w, e_{\mathrm{po}}) \mathrel{-\!\!*} \\ (\text{❷}\mathsf{Po}(\mathsf{dom}(m), e) * \text{❸}\mathsf{FlowSCR}(\Phi, e, x, v, e_w, m, P))\end{array}\right)\end{array}\right\}$$

$$\mathsf{MemRead}\ x$$

$$\{v.\ \exists e, e_w.\ \mathsf{PoPred}(e) * \mathsf{GraphFacts}(e, x, v, e_w, e_{\mathrm{po}}) * \text{❹}e \looparrowright P(x, v, e_w)\}_{tid,\Phi}$$

SCExt-ht-micro-memwrite

$$\left\{\begin{array}{l}\mathsf{PoPred}(e_{\mathrm{po}}) * \mathop{\Large\ast}_{(e_{\mathrm{po}}\mapsto P_{\mathrm{po}})\in m}(e_{\mathrm{po}} \looparrowright P_{\mathrm{po}}) * \\ \forall e.\left(\begin{array}{l}\mathsf{GraphFactsW}(e, x, v, e_{\mathrm{po}}) \mathrel{-\!\!*} \\ (\mathsf{Po}(\mathsf{dom}(m), e) * \mathsf{FlowSCW}(\Phi, e, x, v, m, P))\end{array}\right)\end{array}\right\}$$

$$\mathsf{MemWrite}\ x\ v$$

$$\{().\ \exists e.\ \mathsf{PoPred}(e) * \mathsf{GraphFacts}(e, x, v, e_{\mathrm{po}}) * e \looparrowright P(x)\}_{tid,\Phi}$$

$$\mathsf{FlowSCR}_{\Phi}(e, x, v, e_w, m, P) \triangleq \left(\left(\mathop{\Large\ast}_{(\_\mapsto P_{\mathrm{po}})\in m} P_{\mathrm{po}}\right) * \Phi(x, v, e_w)\right) \Rrightarrow \Phi(x, v, e_w) * P(x, v, e_w)$$

$$\mathsf{FlowSCW}_{\Phi}(e, x, v, m, P) \triangleq \left(\mathop{\Large\ast}_{(\_\mapsto P_{\mathrm{po}})\in m} P_{\mathrm{po}}\right) \Rrightarrow \Phi(x, v, e_w) * P(x)$$

Fig. 20. AxSL$^{\mathrm{SCExt}}$ version of the AxSL$^{\mathrm{SC}}$ rules shown in Fig. 13. The new syntax and changes are highlighted. The user provides m, a thread-local map from events to the resources consumed.

graph reasoning as in the AxSL$^{\mathrm{SC}}$ MP proof. Finally, we have the desired implication tied to event d, which suffices to show the specification.

```
1    ${\mathsf{PoPred}(-) * r_1 \overset{\mathrm{r}}{\mapsto} \_ * r_2 \overset{\mathrm{r}}{\mapsto} \_}$
2       $\forall v, e_w.\ \Phi(flag, v, e_w)$
        $\Rrightarrow$
3       $\Phi(flag, v, e_w) * \Phi(flag, v, e_w)$
4 c: $r_1$ := ldr [$flag$]
5    ${\mathsf{PoPred}(c) * \exists v, e.\ r_1 \overset{\mathrm{r}}{\mapsto} v * c \looparrowright \Phi(flag, v, e) * c{:}\mathrm{R}\ flag\ v * e\ \mathrm{rf}\ c}$
6       $\forall v', e_w.\ \mathsf{GraphFactsR}(d, data, v', c) * \Phi(data, v', e_w) * \Phi(flag, v, e)$
        $\Rrightarrow$
7       $\Phi(data, v', e_w) * (v = 1 \longrightarrow v' = 42)$
8 d: $r_2$ := ldr [$data$]
9    ${\mathsf{PoPred}(d) * \exists v, v'.\ r_1 \overset{\mathrm{r}}{\mapsto} v * r_2 \overset{\mathrm{r}}{\mapsto} v' * d \looparrowright (v = 1 \longrightarrow v' = 42)}$
```

Fig. 21. Proof sketch of the receiving thread of MP in AxSL$^{\mathrm{SCExt}}$. The explicit reasoning of resource flow is highlighted.

5.4 Handling Arm-A Concurrency: The AxSL$^{\mathrm{Arm}}$ Logic

Using AxSL$^{\mathrm{SCExt}}$ for TinySc as the base, we build AxSL$^{\mathrm{Arm}}$ for TinyArm. The changes we make to AxSL$^{\mathrm{SCExt}}$ to obtain AxSL$^{\mathrm{Arm}}$ are twofold. First, we extend the surface assertion language. Similar to how we extend the opax semantics of TinySc to obtain the counterpart of TinyArm, we add new assertions dedicated to reason about Arm's dependencies, etc., and augment existing ones. Second, we change the resource flowing order. We shift from reasoning about resource flow along po to Arm's lob to reflect the change of the synchronisation order from SC's sc to Arm's ob. Arm-A's relaxed concurrency is fundamentally different from SC in the sense that po in Arm-A does not impose an intra-thread ordering, which implies that transferring resources along po is unsound. Thus, in AxSL$^{\mathsf{Arm}}$, we instead rely on lob, the local fragment of which enforces synchronisation. We elaborate on these two changes with the rule for MemRead in §5.4.1.

5.4.1 Proof Rules for Microinstructions. We now explain one of the key proof rules HT-MICRO-MEMREAD-RDEP-EXT in Fig. 22 for a MemRead with ordering strength *os*, variety *vr*, address *x*, and address dependencies *d*, focusing on the new components introduced to handle Arm-A's memory model. 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. 23.

HT-MICRO-MEMREAD-RDEP-EXT

$$\left\{\begin{array}{l} \text{❶}\mathsf{NoLocalWrites}(x) * \text{❷} d = (\mathsf{dom}(regs), \emptyset) * \mathsf{PoPred}(e_{\mathsf{po}}) * \text{❸}\mathsf{CtrlPreds}(srcs_{\mathsf{ctrl}}) * \\ \text{❹} \mathop{\Huge *}_{(r \mapsto (v,E)) \in regs} r \overset{\mathsf{r}}{\mapsto} v@E * \mathop{\Huge *}_{(e_{\mathsf{lob}} \mapsto P_{\mathsf{lob}}) \in m} (e_{\mathsf{lob}} \hookrightarrow P_{\mathsf{lob}}) * \\ \forall e, v, e_w.\ \left(\begin{array}{l} \text{❺}\mathsf{GraphFacts}(e, os, vr, x, v, e_w, e_{\mathsf{po}}, srcs_{\mathsf{ctrl}}, d, regs) \mathrel{-\!\!*} \\ (\text{❻}\mathsf{Lob}(\mathsf{dom}(m), e) * \text{❼}\mathsf{FlowR}(\Phi, e, x, v, e_w, m, P)) \end{array}\right) \end{array}\right\}$$

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

$$\left\{\begin{array}{l} \qquad\quad \exists e, e_w.\ D = \{e\} * \text{❽}\mathsf{NoLocalWrites}(x) * \mathsf{PoPred}(e) * \mathsf{CtrlPreds}(srcs_{\mathsf{ctrl}}) * \\ (v, D).\ \text{❾} \mathop{\Huge *}_{(r \mapsto (v,E)) \in regs} r \overset{\mathsf{r}}{\mapsto} v@E * \mathsf{GraphFacts}(e, os, vr, x, v, e_w, e_{\mathsf{po}}, srcs_{\mathsf{ctrl}}, d, regs) * \\ \qquad\quad e \hookrightarrow P(x, v, e_w) \end{array}\right\}_{tid,\Phi}$$

Fig. 22. A proof rule of AxSL$^{\mathsf{Arm}}$ for the MemRead microinstruction, specialised for a thread that has no writes to the location read. The changes to SCEXT-HT-MICRO-MEMREAD are highlighted.

The first two clauses of the precondition capture the specialisation mentioned above: ❶ $\mathsf{NoLocalWrites}(x)$ 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 ❽. ❷ $d = (\mathsf{dom}(regs), \emptyset)$ requires that this MemRead depends on *exactly* the registers in the domain of the (partial) register file *regs* of ❹ (non-involved registers can be framed off to apply this rule via the normal frame rule familiar from separation logics), and not on any intra-instruction memory read (the $\emptyset$ of event IDs). The collection of register points-to for *regs* of ❹ is unchanged and thus restored in the postcondition as ❾. Note that a register points-to assertion now also maps a register to a set of events E that are the sources of its data, with notation $@E$. This change captures the

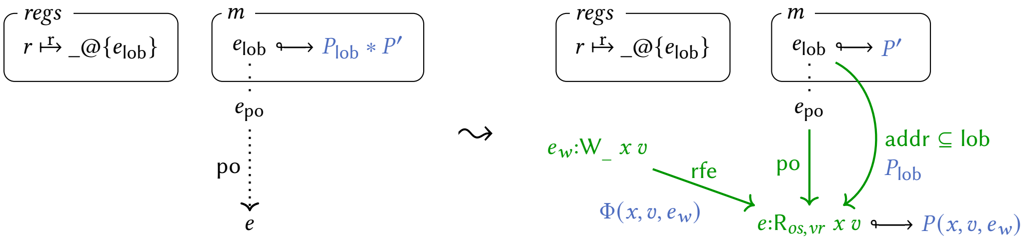

regs
$r \overset{\mathsf{r}}{\mapsto} \_@\{e_{\mathsf{lob}}\}$
m
$e_{\mathsf{lob}} \hookrightarrow P_{\mathsf{lob}} * P'$
e_{po}
po
e
$\rightsquigarrow$
regs
$r \overset{\mathsf{r}}{\mapsto} \_@\{e_{\mathsf{lob}}\}$
m
$e_{\mathsf{lob}} \hookrightarrow P'$
e_{po}
e_w:W_ x v
rfe
po
addr ⊆ lob
P_{lob}
$\Phi(x,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

Fig. 23. A visualisation of HT-MICRO-MEMREAD-RDEP-EXT for external read event e, with logical annotations and newly gained graph facts highlighted. $\rightsquigarrow$ indicates the update of resources from the precondition to the postcondition. The protocol specifies $\Phi(x, v, e_w)$ transferred along the rfe edge to e, which is returned to be transferred to other potential reads. 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.

corresponding extension to the register file in the opax semantics. The GraphFacts predicate in ❺ now takes more arguments, and gives more graph assertions like the dependency edges. Bookkeeping assertion ❸ $\mathsf{CtrlPreds}(srcs_{ctrl})$ captures the $srcs_{\mathsf{ctrl}}$ part of the opax thread state T which works like PoPred. The user-supplied m constrained by $\mathsf{Po}(\mathsf{dom}(m), e)$ in SCExt-ht-micro-memread is now constrained by ❻ Lob which instead requires the domain of m to consist of lob predecessors of the read. The flow equation ❼ and FlowSCR explained in $\text{AxSL}^{\mathsf{SCExt}}$ share the same definition.

The post condition is parametrised by an additional dependency set D (the set of event IDs that v stems from) for intra-instruction dependencies. In this rule, this set is $\{e\}$, where e is the existentially quantified event ID of the associated memory read event, which is passed to the continuation, to, for instance, a register read to establish the dependency from this read event to the register's data.

5.4.2 Specialising Proof Rules for Examples. The Arm-A memory model is intrinsically complex, so there is no 'perfect' rule that is both simple and general. Therefore, we again derive some further specialised instruction rules for reasoning about examples.

We explain how $\text{AxSL}^{\mathsf{Arm}}$ can be used to reason about Arm-A concurrency compositionally, using two key examples: message-passing (§ 5.5) and versions of load buffering (§ 5.6), describing our high-level proof rules along the way. These examples also demonstrate two styles of proof: a low-level proof for (Arm's version of) MP that is similar to that of $\text{AxSL}^{\mathsf{SCExt}}$, showing how one can tackle subtle reasoning about the Arm-A memory model if needed; and a high-level proof for LB that abstracts physical state with simple, easy-to-use ghost state, demonstrating the convenience of $\text{AxSL}^{\mathsf{Arm}}$. For MP, we only sketch the proof of the receiving thread, highlighting the differences to the MP proof done in $\text{AxSL}^{\mathsf{SCExt}}$. For LB, we go through the proof in more detail, and present the specialised rules for the instructions used in the proof along the way, and explain how it fails (as desired) if the necessary synchronisation is removed.

5.5 Graph Reasoning in $\text{AxSL}^{\mathsf{Arm}}$: MP+rel+addr

We give an example illustrating more complex reasoning about the memory event graph, on an Arm-A version of message passing. Comparing to the SC counterpart, two synchronisations are inserted to ensure intra-thread order. The flag write of the sending thread 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 two reads of the receiving thread 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.

$$\begin{array}{l||l} a\colon \mathsf{str}\ [\mathit{data}]\ 42 & c\colon r_1 := \mathsf{ldr}\ [\mathit{flag}] \\ b\colon \mathsf{str}_{\mathsf{rel}}\ [\mathit{flag}]\ 1 & d\colon r_2 := \mathsf{ldr}\ [\mathit{data} + r_1 - r_1] \end{array}$$

Fig. 24. MP+rel+addr: $r_1 = 1 \longrightarrow r_2 = 42$

Overview. The proof has roughly the same shape as the previous ones, in particular we conclude r_2 is 42 with a contradiction on the validity of the graph in the case of reading the initial value 0 from *data* when the flag is set.

Protocol. The protocol we specify is almost identical to the one used for the SC variant, except for two minor adjustments (highlighted in orange) at *flag* due to the now relaxed memory model.

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

First, we additionally require in the right disjunct that the write is from Thread 1 (the sending thread) with $tid(e) = 1$. This allows us to conclude that the load of *flag* on the receiving side has to read from this *external* write, resulting in an rfe edge which contributes to the later graph reasoning because it ensures inter-thread synchronisation (in contrast, rfi in Arm-A does not enforce such a synchronisation). Second, for the same reason, we strengthen the write to *data* to be a release write, which allows us to conclude a lob edge between the two writes of the sending thread.

Proof Sketch. We look at the proof sketch of the receiving thread, focusing on the reasoning of the data dependency between two loads, and the Arm version of the contradiction proof. We learn $r_1 \overset{\mathrm{r}}{\mapsto} v@\{c\}$ after reading from the flag, and $c \hookrightarrow \Phi(\mathit{flag}, v, b)$ for some b. The $@\{c\}$ part of the register points-to tells us that the value v comes from c. Therefore, when we consider the data read, we have c data d because of the artificial data dependency using r_1. The fact that data is in lob allows us 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 $b{:}\mathsf{W}_{\mathsf{rel}}\ \mathit{flag}\ 1$ and $a{:}\mathsf{W}\ \mathit{data}\ 42$ for some a such that a po b. Since e is an initial write to the same address as a, we know e co a, and since e rf d, we know d fr a and therefore d ob a, since d and a are on different threads. Because b is a release write po-after a, we have a ob b; because c reads from b on a different thread (recall the new bits of the protocol), we have b ob c; and finally, because there is a data dependency between c and d, we have c ob d. By transitivity of ob, we obtain a ob d, and together with d ob a, we obtain d ob d, which contradicts the irreflexivity of ob.

5.6 High-Level Reasoning in AxSL$^{\text{Arm}}$: Load Buffering

We detail how to obtain a high-level proof of LB+artificialdata+data (Fig. 25), 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 proven using just 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. 25 specification. We give two versions of the postcondition: one still involving tied assertions, corresponding to the final state of the threads, and one where the assertions have been pulled out, as used in our formal definition of weakest preconditions, as we describe in §6.5.5.

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

Fig. 25. 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 $\mathsf{shot}(v)$ represents the information that the decision has been made to choose value v. In particular, $\boxed{\mathsf{pending}} * \boxed{\mathsf{pending}}$ is a contradiction, and so is $\boxed{\mathsf{pending}} * \boxed{\mathsf{shot}(v)}$, but we can view-shift pending into $\mathsf{shot}(v)$, $\boxed{\mathsf{pending}} \Rrightarrow \boxed{\mathsf{shot}(v)}$, 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 $\boxed{\mathsf{shot}(1)}$.

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

Thread 1:

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

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

$\Rrightarrow$

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

4 a: $r_1 := \mathsf{ldr}\ [x]$

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

6 $\boxed{\mathsf{pending}}$

$\Rrightarrow$

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

8 b: $\mathsf{str}\ [y]\ (1 + r_1 - r_1)$

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: $r_2 := \mathsf{ldr}\ [y]$

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

13 d: $\mathsf{str}\ [x]\ r_2$

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

Fig. 26. Proof sketch of LB+artificialdata+data.

Proof sketch. Using this protocol, the proof follows the sketch in Fig. 26 using the specialised instruction proof rules explained later in §5.6.1. 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 in line 4 to show that it must read 0. [4]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 $\mathsf{shot}(v)$; 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, 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.

[4] 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 do not imply ob on Arm-A, and thus has a significantly weaker premise for its flow implication, without $\Phi(x, v_1, \_)$.

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.

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.

5.6.1 *Proof Rules for Instructions.*

We explain the specialised instruction proof rules used in the LB proof, focusing on (highlighted in orange) how the rules for Arm-A (Fig. 27) differ from the (non-specialised) rules for SC (Fig. 15).

Load. The proof rule that we use for the load in both threads, ht-ins-ldr-pln-ext, 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, the program counter, and the local writes. The second 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 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

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

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

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

HT-INS-STR-PLN-ARTIFICIALDATA

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

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

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

Fig. 27. Proof rules for the instructions in the left thread of LB+artificialdata+data.

5.7 Supporting Exclusives

We describe how the logic we have seen so far can be extended, with only minor changes, with new proof rules for Arm-A 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 'non-exclusive' 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 which does not need to be given back because of the exclusivity.

```
trylock(ℓ) ≜
  r₁ := ldrₓ [ℓ]
  if (r₁ ≠ unlocked) return false
  r₂ := strₓ [ℓ] locked
  dmb sy
  return (r₂ = success)

unlock(ℓ) ≜ str_rel [ℓ] unlocked
```

Fig. 28. Try-lock pseudocode

```
if trylock(ℓ) {   ‖ if trylock(ℓ) {
  str [x] 1       ‖   r₁ := ldr [x]
  str [y] 1       ‖   r₂ := ldr [y]
  unlock(ℓ)       ‖   unlock(ℓ)
}                 ‖ }
```

Fig. 29. Example of mutual exclusion with postcondition $r_1 = r_2$

To support atomic read-modify-writes, we do not need to change our definition of protocols, but merely to pick a specific shape of protocol. The protocol relies on invariants, which we implement 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 $\mathsf{ExTok}(e)$ to each event e, and to make that token available to the rule for that event. We implement this idea in §6.5.4.

Using our proof rules, we prove the classical specification for simple try-lock (see Fig. 28), which we then use to prove a basic mutual exclusion example (in Fig. 29): 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.

5.8 Further Examples

To validate how AxSL$^{\mathrm{Arm}}$ 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 (Fig. 30) 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 §5.6. LB+ctrls (Fig. 31) relies on the `ctrl; [W]` clause of dob in both threads, but is otherwise the same. Similarly, MP+rel+dmbsy (Fig. 32) relies on bob in the right thread, and so does MP+rel+ctrl-isb (Fig. 33) via the more complex `(ctrl|(addr; po)); [ISB]; po; [R]` edge which appears incrementally, but their proofs are otherwise as in §5.5. To illustrate that AxSL$^{\mathrm{Arm}}$ can reason about communication between many threads, we verify an iterated version of MP, namely ISA2+rel+data+acq (Fig. 34) [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 §8.3), we verify two coherence tests: CoWW (Fig. 35) and CoRR (Fig. 36); the proofs work by symbolic execution followed by discarding the executions with cycles in co.

```
a: r1 := ldr [x]  ║ d: r2 := ldr [y]
b: dmb sy         ║ e: str [x] r2
c: str [y] 1      ║
```

Fig. 30. LB+dmbsy+data: $r_1 = 0 \land (r_2 = 0 \lor r_2 = 1)$

```
a: r1 := ldr [x]   ║ d: r2 := ldr [y]
b: if (r1 == 0)    ║ e: if (r2 == 1)
c:     str [y] 1   ║ f:     str [x] r2
```

Fig. 31. LB+ctrls: $r_1 = 0 \land (r_2 = 0 \lor r_2 = 1)$

```
a: str [data] 42     ║ c: r1 := ldr [flag]
b: str_rel [flag] 1  ║ d: dmb sy
                     ║ e: r2 := ldr [data]
```

Fig. 32. MP+rel+dmbsy: $r_1 = 1 \longrightarrow r_2 = 42$

```
a: str [data] 42     ║ c: r1 := ldr [flag]
b: str_rel [flag] 1  ║ d: if (r1 == 1)
                     ║ e:     isb
                     ║ f:     r2 := ldr [data]
```

Fig. 33. MP+rel+ctrl-isb: $r_1 = 1 \longrightarrow r_2 = 42$

```
a: str [x] 42     ║ c: r1 := ldr [y]  ║ e: r2 := ldr [z]
b: str_rel [y] 1  ║ b: str [z] r1     ║ d: r3 := ldr [x]
```

Fig. 34. ISA2+rel+data+acq: $r_2 = 1 \longrightarrow r_3 = 42$

```
a: str [x] 37
b: str [x] 42
```

Fig. 35. CoWW: $a \xrightarrow{\mathrm{co}} b$

```
a: str [x] 42  ║ b: r1 := ldr [x]
               ║ c: r2 := ldr [x]
```

Fig. 36. CoRR: $r_1 = 42 \longrightarrow r_2 = 42$

6 MODEL AND SOUNDNESS

A model of Hoare triples for a language of microinstructions is said to be sound if the following two conditions hold: (1) the model must allow us to show that proof rules are valid with respect to the language semantics (soundness), i.e. that the transformation of logical resources reflects the transition of physical states; (2) we must be able to show that, if a program is verified against a Hoare triple, then all valid executions of the program indeed satisfy certain properties (adequacy).

With these two principles in mind, we present the semantic models of the three logics presented in the previous section. We also sketch how to prove the soundness of proof rules using the models, and briefly touch on how the model definitions contribute to adequacy (we leave the details to §7).

Following the structure of the previous section, we build up the model of AxSL$^{\mathsf{Arm}}$ in Iris gradually with several intermediate steps, tackling one challenge at a time.

First, we recall the model of usual Iris logics built atop of a heap-based operational semantics, in a setting with SC concurrency and a fixed number of threads. We emphasise some core components of the model that are crucial for soundness and adequacy.

Next, we sketch the model of AxSL$^{\mathsf{SC}}$ based on our opax semantics for TinySc. We explain how it deals with the shape of opax semantics to achieve modular graph-based reasoning, and give an intuitive explanation on why models as such do not scale to the relaxed concurrency of Arm.

Then, we present the model of AxSL$^{\mathsf{SCExt}}$ which is fundamentally distinct, due to our now explicit treatment of resource flowing. We concentrate on the flow implications, on how it works together with tied-to assertions to ensure soundness, and why this idea scales to relaxed concurrency.

Finally, we show the full semantic model of AxSL$^{\mathsf{Arm}}$, which shares the same structure as that of AxSL$^{\mathsf{SCExt}}$, but differs due to the shift of the synchronisation order from SC's sc to Arm's ob. We omit the soundness proof of the proof rules of AxSL$^{\mathsf{Arm}}$ as it is mostly orthogonal to the memory model, thus similar to the one for AxSL$^{\mathsf{SCExt}}$.

This section presupposes some basic knowledge of Iris. See [Jung et al. 2018] for background on Iris.

6.1 Preliminary: A General Recipe for Building Logics Using Iris

Iris can be instantiated by an operational semantics respecting a few conditions. The framework comes with a recipe for building logics and their adequacy proofs for typical heap-based operational semantics. We now recall this recipe, which we refer to as the *general recipe*. Readers who are familiar with this general recipe may skip this subsection.

Base operational semantics. To instantiate Iris, one normally needs to provide a concurrent language whose semantics has:

- a set of expressions Exp, and values $v : \mathsf{Val} \subseteq \mathsf{Exp}$
- a notion of physical state $\sigma : \mathsf{St}$ shared among all threads
- a per-thread step relation $\rightarrow_{tid} \subseteq (\mathsf{Exp} \times \mathsf{St}) \times (\mathsf{Exp} \times \mathsf{St})$ that specifies how the program of thread *tid* may transform the shared state for a primitive step

For simplicity, we only consider a language with a fixed number of threads, i.e. without the ability to fork new threads.

Weakest precondition. Given such a semantics, one can follow the general recipe to first define a notion of weakest precondition. Here, we only show the key parts of the definition, and we refer the reader to [Jung et al. 2018] for a full definition that supports concurrency. Intuitively, $wp\ e\ \{Q\}_{tid}$ requires post condition Q to hold if e terminates with a value in thread *tid*. A Hoare triple $\{P\}\ e\ \{Q\}_{tid}$ is then defined as $\Box(P \twoheadrightarrow wp\ e\ \{Q\}_{tid})$. The definition of weakest precondition has two cases, depending on whether the program e is a value:

- If e is a value, the weakest precondition requires that the postcondition holds after a resource update: $\dot{\Rrightarrow} Q(e)$

- Otherwise, for all physical updates from global state s, the logical resources are required to be updated to mirror them, using a *state interpretation* (SI) which is a predicate that gives the physical state a logical interpretation in Iris:

  $$\mathsf{SI}(s) \Rrightarrow (\forall e', s'.\, (e, s) \rightarrow_{tid} (e', s') \Rrightarrow \mathsf{SI}(s') * wp\; e'\; \{Q\}_{tid})$$

  Formally, it says that for any physical transition $(e, s) \rightarrow_{tid} (e', s')$ of thread *tid*, there must be a corresponding logical transition formulated as a view shift $\mathsf{SI}(s) \Rrightarrow \mathsf{SI}(s')$. The recursive occurrence of the weakest precondition further requires this correspondence to hold for all subsequent steps of the thread. (Here we omit substantial technical details dealing with invariants and guarded recursion.)

State interpretation. Normally, to achieve local reasoning, one would define SI as an authoritative view of the physical state s, and only reason about fragmental views distributed to threads, separating s so that one does not need to reason about the parts of s that are not touched by a thread (known as framing). For instance, for an SC language with a global abstract heap (a map from locations to values) s, one can define the full view over s as $\mathsf{SI}(s) \triangleq \boxed{\bullet s}$ and a fragmental view for an individual location l as $l \mapsto v \triangleq \boxed{\circ\{l \mapsto v\}}$ using the *Auth* ghost state constructor of Iris, such that (1) the two views are consistent: $\mathsf{SI}(s) * l \mapsto v \vdash s(l) = v$, and (2) we can update them together: $\mathsf{SI}(s) * l \mapsto v \Rrightarrow \mathsf{SI}(s[l \mapsto v']) * l \mapsto v'$.

Adequacy. Next, we show the statement of a typical adequacy theorem, and highlight the role of the state interpretation SI in the proof of the theorem. The theorem has the following notable assumptions (again, we omit substantial details):

- a thread pool step relation $\rightarrow_{\mathrm{tp}} \subseteq (\mathrm{list}(\mathsf{Exp}) \times \mathsf{St}) \times (\mathrm{list}(\mathsf{Exp}) \times \mathsf{St})$ that specifies the scheduling of threads — all possible interleavings of primitive steps,
- a terminating thread pool trace $([e_0, \cdots, e_n], s_i) \rightarrow^{*}_{\mathrm{tp}} ([e'_0, \cdots, e'_n], s_t)$ where s_i is the initial state, s_t is the terminating state, and all e's are values,
- a series of weakest preconditions with post conditions $\phi_0, \ldots, \phi_n$, one for each thread

The conclusion is that the post conditions $\phi_i(e'_i)$ for all i hold. This theorem allows one to extract meta-logical results from the program logic, showing that verification done in the program logic is valid with respect to the meta logic.

Proof of adequacy. The proof of adequacy proceeds by allocating the initial logical interpretation $\mathsf{SI}(s_i)$ to establish the physical-logical correspondence for s_i. We then continue by induction on the trace to show that the correspondence is preserved throughout the execution. In the induction case, when we have a head step $(e_i, s) \rightarrow_{tid} (e'_i, s')$ for thread *tid* and $\mathsf{SI}(s)$, we unfold the definition of weakest precondition of thread i, which gives us $\mathsf{SI}(s) \Rrightarrow \mathsf{SI}(s')$. Since we own $\mathsf{SI}(s)$, we apply the view shift to obtain $\mathsf{SI}(s')$. The case is concluded by applying the induction hypothesis which requires $\mathsf{SI}(s')$ and the remaining reduction steps. In the end, we reach the final state s_t, at which point we get the post condition by the value case of the weakest precondition.

Two orders. A crucial observation about this proof that will become relevant when we adapt the general recipe to work with an opax semantics is that the proof implicitly works with two orders: the order that the physical state evolves in, and the order in which the transformations of the corresponding logical interpretations are collected by weakest preconditions. The first order is induced by the thread pool reduction $\rightarrow_{\mathrm{tp}}$ of the semantics, which is a serialisation of all accesses to the physical state (from the perspective of the axiomatic model, a linear extension of sc). The second order is program order. From the model of the weakest precondition, we can see that a view shift is required for each

$$\begin{tikzcd}
(e_{0,0}, s_0) \arrow[d, "0"'] & \mathsf{SI}(s_0) \arrow[d, Rrightarrow] & & \mathsf{SI}(s_2) \arrow[d, Rrightarrow] & (e_{1,0}, s_2) \arrow[d, "1"] \\
(e_{0,1}, s_1) \arrow[d, "0"'] & \mathsf{SI}(s_1) \arrow[rru, Rightarrow] & & \mathsf{SI}(s_3) \arrow[lld, Rightarrow] & (e_{1,1}, s_3) \arrow[d, "1"] \\
(e_{0,2}, s_4) & \mathsf{SI}(s_4) \arrow[rr, Rightarrow] & & \mathsf{SI}(s_5) & (e_{1,2}, s_5)
\end{tikzcd}$$

Fig. 37. An example program execution with two threads, above an SC memory model. $\rightarrow_0$ and $\rightarrow_1$ are the head reductions (the program order and the reasoning order) of the two threads. The indices of s indicate the global order of updating s (induced from $\rightarrow_{\mathsf{tp}}$), and $\Rrightarrow$ indicates the order we update the logical interpretations of s in the adequacy proof. These two orders coincide, and they subsume the reasoning order.

local reduction $\rightarrow_{tid}$ of the semantics. Fig. 37 illustrates the relationship between these two orders. More generally, we observe that, in most logics, the second order is program order, since it is intuitive to reason about programs along program order, whereas the first order may vary depending on the concurrency model.

Now, a crucial observation is the following: the reason we can complete the proof in one pass (with one induction), with a single induction on the former order, is that in the case of SC, the second order is included in the first: po $\subseteq$ sc. As discussed in §3, in the case of Arm, these two orders are incompatible, in the sense that their union may be cyclic, and so they together do not form an inductive structure that we can rely on in the proof. This poses a challenge to the adequacy proof, and partially explains why the general recipe does not work for the relaxed concurrency of Arm.

6.2 Notations for Weakest Preconditions

In the rest of this section, we discuss variants of weakest preconditions, for our three logics at two abstraction levels. We now clarify the convention of notations we will use for them. We will use superscripts SC, SCExt, and Arm to denote that assertions we are talking about belong to AxSL$^{\mathsf{SC}}$, AxSL$^{\mathsf{SCExt}}$, and AxSL$^{\mathsf{Arm}}$ respectively.

6.3 Model and Soundness of AxSL$^{\mathsf{SC}}$

AxSL$^{\mathsf{SC}}$ has a simple semantic model and a one-pass adequacy proof similar to most Iris program logics. AxSL$^{\mathsf{SC}}$ shares the exact core idea of updating logical resources respecting the transformation of physical states as in the general recipe. Following the recipe, we let the thread state of the opax semantics s be the expression; the complete state Done $\langle\_\rangle$ be the value; the pair of execution graph and instruction memory $\langle X, I\rangle$ be the physical state σ; and $\xrightarrow{tid}_{\mathsf{h}}$ be the per-thread step. The notation $s \xrightarrow{tid,X,I}_{\mathsf{h}} s'$ used in Fig. 11 is syntactic sugar for $\langle s, \langle X, I\rangle\rangle \xrightarrow{tid}_{\mathsf{h}} \langle s', \langle X, I\rangle\rangle$. With this configuration, we define a *base weakest precondition* $\mathsf{wpb}^{\mathsf{SC}}_{tid,\Phi}\ s\ \{Q\}$ that takes an opax state s (the "expression") and a post condition Q which is a predicate over the terminating state, in the spirit of the aforementioned general recipe, in Fig. 38.

A clearer interface. On top of $\mathsf{wpb}^{\mathsf{SC}}$, which we will explain soon, we define the weakest precondition $\mathsf{wp}^{\mathsf{SC}}_{tid,\Phi}\ p\ \{Q\}$ that merely takes a microinstruction program p, as a cleaner interface hiding s away. We implement this using usual Iris machinery:

$$\mathsf{wp}^{\mathsf{SC}}_{tid,\Phi}\ p\ \{Q\} \triangleq \forall T.\ \text{❶}\mathsf{LSI}(T)_{tid} \Rrightarrow \mathsf{wpb}^{\mathsf{SC}}_{tid,\Phi}\ \mathsf{Ctd}\ \langle p, T\rangle\ \Big\{T'.\ Q * \text{❷}\mathsf{LSI}(T')_{tid}\Big\}_{tid,\Phi}$$

We have a *local state interpretation* ❶ $\mathsf{LSI}(T)_{tid}$ interpreting T for the current thread *tid*. LSI relates (local) logical assertions to the corresponding part of T. It allows us to universally quantify T and use the assertions to only track the parts that are necessary for further reduction from the opax state. The ❷ $\mathsf{LSI}(T')_{tid}$ in the post condition of the $\mathsf{wp}^{\mathsf{SC}}$ requires a new interpretation for the updated local state T'. Note that it is a thread-local predicate which only involves assertions of a thread (indexed by a thread ID) (for Iris experts, thread-local assertions use distinct ghost names).

$$
\begin{aligned}
\mathsf{wpb}^{\mathsf{SC}}_{tid,\Phi}\, s\, \{Q\} \triangleq{} & \big(s = \mathsf{Done}\, T \wedge \mathrel{|\!\dot{\Rrightarrow}} Q(T)\big) \vee \\
& \Bigg(s = \mathsf{Ctd}\, C \wedge \\
& \Bigg(\forall \sigma.\, \text{❸}\mathsf{Valid}(\sigma.X) \mathrel{-\!\!*} \text{❹}(\Box\, \mathsf{SI}(\sigma)) \mathrel{-\!\!*} \forall s'.\, \text{❺}\langle s, \sigma\rangle \xrightarrow{tid}_{\mathsf{h}} \langle s', \sigma\rangle \mathrel{-\!\!*} \\
& \quad \forall e = \langle tid, C.T.IT.cntr\rangle, \big(\mathsf{IsValidEid}(e, X) * \text{❻}s_{pg} \supseteq \mathsf{Sc}(\sigma.X, e) \wedge e \notin s_{pg}. \\
& \qquad \text{❼}\mathsf{SIP}(\Phi, \sigma.X, s_{pg}) \Rrightarrow \mathsf{SIP}(\Phi, \sigma.X, s_{pg} \cup \{e\}) * \text{❽}\mathsf{wpb}^{\mathsf{SC}}_{tid,\Phi}\, s'\, \{Q\}\big) \\
& \qquad \vee (\text{❾}\neg \mathsf{IsValidEid}(e, X) * \mathsf{wpb}^{\mathsf{SC}}_{tid,\Phi}\, s'\, \{Q\})\Bigg)\Bigg)
\end{aligned}
$$

Fig. 38. The model of $\mathsf{wpb}^{\mathsf{SC}}$. Technical details regarding guarded recursion are omitted.

Overall structure. Following the general recipe, the definition of $\mathsf{wpb}^{\mathsf{SC}}$ has two cases, depending on whether s is a "value". In the value case, we just get the local state T and assert post condition $Q(T)$ after a ghost update. In the other case, there are two aspects of the definition which are somewhat non-standard compared to the general recipe: (1) it maintains consistency between the execution of the thread in the opax semantics and the global execution graph, to ensure sound graph reasoning; (2) it enforces rely-guarantee protocols on the graph to implement resource transfer.

Persistent graph as memory. Having an ongoing execution $\mathsf{Ctd}\ C$, we obtain assertions over the "physical state" σ. However, unlike in the recipe when we just assume $\mathsf{SI}(\sigma)$ for heap σ, now we have ❹ $\Box\,\mathsf{SI}(\sigma)$. It is a *persistent* interpretation of σ, a pair of execution graph (the "shared memory") and instruction memory, reflecting the fact that in opax both of them are constant. Additionally, ❸ $\mathsf{Valid}(\sigma.X)$ assumes the validity condition of the execution graph, which helps us rule out ill-formed or inconsistent graphs — we discard a graph when we obtain information from the semantics that conflicts with this assumption (as demonstrated at the logic level in examples in §5). Next, as in the general recipe, we assume that we can make progress by taking a per-thread step (❺), which updates the state to s'. The weakest precondition should hold recursively for s' (as ❽) — the weakest precondition predicate is defined as the (guarded) fixed point satisfying the recursive equation. [5] We also need a case distinction using $\mathsf{IsValidEid}$ to check whether the event ID e corresponds to an event. In the case when we run out of microinstructions and have to reload (the microinstruction program of) the next instruction (namely e is invalid, as ❾), we simply proceed with s'. This case disctinction is just our ad-hoc way of handling the reloading step of opax, and there might be other more systematic solutions.

Enforcing protocols. Finally, we have to show that for the current microinstruction (the first one in $C.p$), the associated event with ID e preserves the protocol Φ for the rely-guarantee style reasoning. Concretely, we use the *progress set* s_{pg}, a set of event IDs, to track how far we are from enforcing the protocol on all nodes of the guessed graph $\sigma.X$, and from checking the consistency between the guessed graph $\sigma.X$ and the program. It contains the set of the events that have been confirmed to conform the protocol and have corresponding microinstructions. We need to show that for an s_{pg} that contains (at least) all sc predecessors but not e, as assumed by ❻, the protocol holds on e (so we make

[5] Our definition does not require that s can take a step in that case; thus, our definition does not enforce progress.

progress by adding e) given it holds on all events in s_{pg} (as ⑦), as illustrated in Fig. 39. The predicate $\mathsf{SIP}(\Phi, X, s_{pg})$ is an interpretation of the progress set given a protocol Φ, which enforces the protocol on all write events in s_{pg}, as captured by:

$$\mathsf{SIP}(\Phi, X, s_{pg}) \triangleq \mathop{\Large\ast}_{e \in s_{pg}} \forall x, v.\ (X.\mathsf{lab}(e) = \mathsf{W}\ x\ v) \Rrightarrow \Phi(x, v, e)$$

The view shift ⑦ means that we can *rely* on all sc-before events (i.e. those that are visible to the current event) conforming the protocol to *guarantee* that e also conforms the protocol. This view shift is crucial for proving the soundness of resource transfer happening in the proof rules. In particular, in the case of e being a read event, $s_{pg} \supseteq \mathsf{Sc}(X, e)$ of ⑥ ensures that the protocol holds at all possible writes that it may read from (since they are all sc-before), which allows us to transfer the protocol resource along the rf from the actual write to this read. In the case of e being a write event, $\mathsf{SIP}(\ldots, s_{pg} \cup \{e\})$ requires the protocol resource of e.

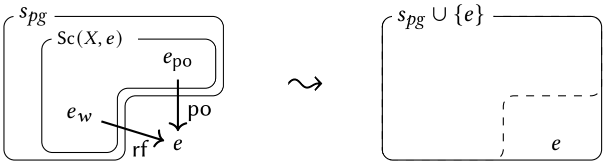


Fig. 39. An illustration on the evolving of the progress set s_{pg} at a read event e.

6.3.1 *Soundness of Proof Rules.* In the elaboration of the model above, we have intentionally left the precise definition of the predicates LSI and SI undefined, since their implementations are irrelevant to the model (and to the adequacy proof), and only pertain to the soundness of the proof rules. Indeed, to show soundness of the proof rules for AxSL$^{\mathsf{SC}}$, it suffices that the predicates satisfy the selection of properties in Fig. 40. We refer readers to Appendix C for a concrete definition of the predicates that satisfies all these properties.

LSI-REG-AGREE

$$\mathsf{LSI}(T)_{tid} * r \overset{\mathsf{r}}{\mapsto} v \vdash T.regs(r) = v$$

LSI-REG-UPDATE

$$\frac{T'.regs = T.regs[r \mapsto v']}{\mathsf{LSI}(T)_{tid} * r \overset{\mathsf{r}}{\mapsto} v \Rrightarrow \mathsf{LSI}(T')_{tid} * r \overset{\mathsf{r}}{\mapsto} v'}$$

LSI-PO-AGREE

$$\mathsf{LSI}(T)_{tid} * \mathsf{PoPred}(e) \vdash \langle tid, T.IT.cntr\rangle > e$$

LSI-PO-UPDATE

$$\frac{\langle tid, T'.IT.cntr\rangle > e'}{\mathsf{LSI}(T)_{tid} * \mathsf{PoPred}(e) \Rrightarrow \mathsf{LSI}(T')_{tid} * \mathsf{PoPred}(e')}$$

SI-EDGE-AGREE

$$\frac{\mathsf{Rel}\text{ is a relation}}{\mathsf{SI}(\sigma) * a\ \mathsf{Rel}\ b \vdash (a, b) \in \sigma.X.\mathsf{Rel}}$$

SI-EDGE-ALLOC

$$\frac{\mathsf{Rel}\text{ is a relation} \qquad (a, b) \in \sigma.X.\mathsf{Rel}}{\mathsf{SI}(\sigma) \vdash a\ \mathsf{Rel}\ b}$$

Fig. 40. Selected properties of state interpretation and local state interpretation. Generally speaking, for mutable fields of T, we require agreement and update rules, while for the immutable graph X, we do not require an update rule.

Single-step weakest preconditions. Weakest preconditions specify the behaviours of a whole microinstruction program execution, while we need a mechanism to specify the behaviours of a single microinstruction to formulate the proof rules for it. Inspired by previous verification work for low-level languages [Erbsen et al. 2021; Jensen et al. 2013; Liu et al. 2023; Myreen and Gordon 2007], we use *single-step* base weakest precondition $\mathsf{sswpb}^{\mathsf{SC}}_{tid,\Phi}\ s\ \{s'.\ Q\}$ which means that Q holds after taking one reduction step (namely executing one microinstruction) from state s. We define $\mathsf{sswpb}^{\mathsf{SC}}$ simply by replacing the recursive occurrence of $\mathsf{wpb}^{\mathsf{SC}}$ with Q in the definition of $\mathsf{wpb}^{\mathsf{SC}}$. The single-step base weakest precondition formally corresponds to a single unfolding of the base weakest precondition, as captured by:

SSWPB-WPB

$$\mathsf{sswpb}^{\mathsf{SC}}_{tid,\Phi}\ s\ \left\{s'.\ \mathsf{wpb}^{\mathsf{SC}}_{tid,\Phi}\ s'\ \{Q\}\right\} \dashv\vdash \mathsf{wpb}^{\mathsf{SC}}_{tid,\Phi}\ s\ \{Q\}$$

Crucially, the right-to-left direction allows us to decompose a microinstruction program to only focus on one microinstruction at a time. This essentially plays the role of the bind rule in logics for high-level languages.

Further, we define the single-step weakest precondition $\mathsf{sswp}^{\mathsf{SC}}\ p\ \{Q\}_{tid,\Phi}$, and finally define the microinstruction Hoare triple used by the proof rule for microinstruction i as

$$\{P\}\ i\ \{Q\}_{tid,\Phi} \triangleq \Box\left(\forall p.\ (P * \exists K.\ (\mathsf{Next}\ i\ K) = p) \mathrel{-\!\!*} \mathsf{sswp}^{\mathsf{SC}}_{tid,\Phi}\ p\ \{Q\}\right)$$

where we require that the first microinstruction of p is i.

We use single-step microinstruction Hoare triples to specify proof rules, and then show the soundness result of AxSL$^{\mathsf{SC}}$:

Theorem 6.1. *The AxSL^{SC} proof rules for microinstructions are sound.*

Proof sketch. We describe the general approach for a proof of soundness of a proof rule here. There are four major steps in the proof after unfolding the model of assertions. First, we use the agreement rules (Fig. 40) between the assertions in the precondition and the interpretation to partially recover the state s. Next, we take the opax step for the microinstruction, obtaining new facts about the execution graph and an updated state s'. At the same time, we perform resource transfer according to the protocol Φ. Finally, we allocate and update assertions to mirror the update of the local state (again using the LSI and SI rules in Fig. 40) and the resource transfer. □

6.4 Model and Soundness of AxSL$^{\mathsf{SCExt}}$

The AxSL$^{\mathsf{SCExt}}$ logic extends the assertion language of AxSL$^{\mathsf{SC}}$ with tied-to assertions, which make it possible to reason about resource flowing between nodes. As a logic also built atop of an opax shape semantics, its model shares substantial similarities with that of AxSL$^{\mathsf{SC}}$. The key difference is how it keeps track of all tied-to assertions to enable sound transfer of tied resources between events. As noted, explicitly tracking resource transfer between events does not add more expressive power to AxSL$^{\mathsf{SCExt}}$, but makes it more general in the sense that both its model and adequacy result are robust for more (relaxed) concurrency models. This is reminiscent of how invariants are tracked in the weakest precondition for the standard Iris program logic [Jung et al. 2018]. We present the model in Fig. 41 and highlight how the model manages tied-to assertions below.

Interpretation for tied resources. SI_{T} of ❶ interprets the full authoritative view of all tied assertions, τ, which we keep as a logical map from event IDs to Iris propositions. Since τ mentions Iris propositions and itself is interpreted as an Iris proposition, we leverage Iris' support for higher-order ghost states to implement SI_{T}. The predicate SI_{T} is defined in such a way that, together with the fragmental tied-to-assertions, it enjoys agreement and update rules similar to those of the register map shown before. (For Iris experts we remark that the rules are slightly different since τ is higher-order: specifically, we can only obtain the agreement *later*.) The next line essentially allows us to update a fragment of τ (to τ',

$$
\begin{aligned}
\mathsf{wpb}^{\mathsf{SCExt}}_{tid,\Phi}\ s\ \{Q\} \triangleq\ & \big(s = \mathsf{Done}\ T \wedge \dot{\Rrightarrow} Q(T)\big) \vee \\
& \left(\begin{aligned} & s = \mathsf{Ctd}\ C \wedge \\
& \left(\begin{aligned} & \forall \sigma.\ \mathsf{Valid}(\sigma.X) \mathrel{-\!\!*} (\Box\, \mathsf{SI}(\sigma)) \mathrel{-\!\!*} \forall s'.\ \langle s, \sigma\rangle \xrightarrow{tid}_{\mathsf{h}} \langle s', \sigma\rangle \mathrel{-\!\!*} \\
& \quad \forall e = \langle tid, C.T.IT.cntr\rangle, \tau.\ \big(\mathsf{IsValidEid}(e, X) * \textcircled{1}\mathsf{SI}_{\mathsf{T}}(\tau) \Rrightarrow \\
& \qquad \exists \tau'.\ \textcircled{2}\mathsf{SI}_{\mathsf{T}}(\tau') * \textcircled{3}\mathsf{FlowImp}(\sigma.X, \Phi, e, \tau, \tau') * \mathsf{wpb}^{\mathsf{SCExt}}_{tid,\Phi}\ s'\ \{Q\}\big) \\
& \qquad \vee(\neg \mathsf{IsValidEid}(e, X) * \mathsf{wpb}^{\mathsf{SCExt}}_{tid,\Phi}\ s'\ \{Q\})
\end{aligned}\right)
\end{aligned}\right) \\
\mathsf{FlowImp}(e, X, \Phi, \tau, \tau') \triangleq\ & \exists \tau_{in}, \tau_{res}, R.\ \textcircled{4}\mathsf{Detach}(X, e, \tau, \tau_{in}, \tau_{res}) * \textcircled{5}\tau' = \tau_{res}[e \mapsto R] * \\
& \forall s_{pg} \supseteq \mathsf{PredOf}(X.\mathsf{sc}, e) \wedge e \notin s_{pg}. \\
& \quad \textcircled{6}\Big(\mathop{\Large *}_{(e \mapsto R_{in}) \in \tau_{in}} R_{in}\Big) * \mathsf{SIP}(\Phi, X, s_{pg}) \Rrightarrow \mathsf{SIP}(\Phi, X, s_{pg} \cup \{e\}) * \textcircled{7}R
\end{aligned}
$$

Fig. 41. The model of base weakest precondition of AxSL$^{\mathsf{SCExt}}$. The key changes to that of AxSL$^{\mathsf{SC}}$ are highlighted in orange. Again, the details handling guarded recursion are omitted.

as $\textcircled{2}$) and the associated tied assertions. Crucially, the update has to follow the flow implication $\textcircled{3}$ FlowImp, which we now explain.

Flow implication. The $\mathsf{FlowImp}(X, \Phi, e, \tau, \tau')$ predicate regulates the flow and update of resources (from τ to τ') that may happen at memory event e. Intuitively, the rule expresses that the sum of resources pushed to e along its incoming sc edges implies (with a view shift) the resources given out along outgoing sc edges, plus the leftovers tied to e.

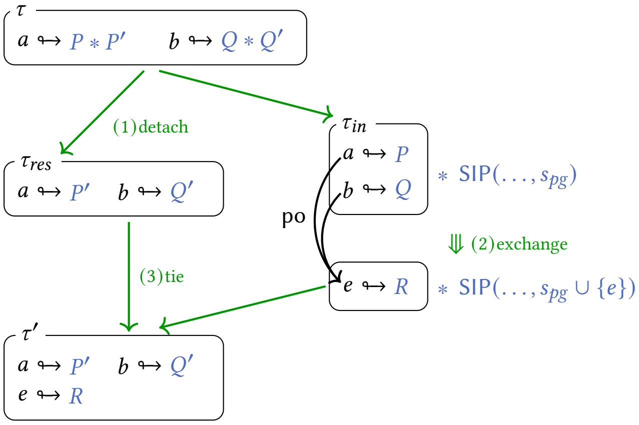


Fig. 42. A visualisation of the three actions of updating τ to τ', where we flow P and Q to e along po from a and b respectively. R is eventually tied to e after exchange.

We divide the update from resource map τ to τ' into a sequence of three actions: detach, exchange, and tie, as illustrated in Fig. 42. The $\textcircled{4}$ Detach predicate captures the detachment:

$$
\begin{aligned}
\mathsf{Detach}(X, e, \tau, \tau_{in}, \tau_{res}) \triangleq\ & \mathsf{dom}(\tau_{in}) \subseteq \mathsf{PredOf}(X.\mathsf{po}, e) * \forall (e \mapsto R_{in}) \in \tau_{in}. \\
& \exists R_{res} = \tau_{res}(e), R = \tau(e).\ R \mathrel{-\!\!*} (R_{in} * R_{res})
\end{aligned}
$$

τ is split into τ_{in} and τ_{res}, where τ_{in} is a fragment of τ which represents the resources *detached* from the po predecessors of e, as determined by the user-provided tied assertions, and τ_{res} is the remaining global map, after detaching τ_{in}. [6] The last two lines do the resource *exchange*, which is an augmented version of the view shift presented in the model of $\mathrm{AxSL}^{\mathsf{SC}}$. It now has the local resources of τ_{in} explicitly given on the left as ⑥, and the resource ⑦ R remaining on event e on the right. The map update ⑤ does the tying: the remaining resource R is *tied* to e, by extending the map τ_{res}.

6.4.1 Stratification. The key difference between $\mathrm{AxSL}^{\mathsf{SC}}$ and $\mathrm{AxSL}^{\mathsf{SCExt}}$ is the tied-to assertions: $a \looparrowright P$. The intent of $a \looparrowright P$ is to express that P holds at event a, which is po-before the current event (if we ignore the case where P is transferred to another thread). In $\mathrm{AxSL}^{\mathsf{SCExt}}$, this then allows us to transfer P to the current event, or to any future event of the thread, and so $a \looparrowright P$ amounts exactly to P. However, when we consider an actual relaxed memory model in $\mathrm{AxSL}^{\mathsf{Arm}}$, this transfer step will be conditional on ordering (in lob) from a to the current event, and the indirection induced by $a \looparrowright P$ allows us to express that P is available under this condition.

The effect of tying P to a in $\mathrm{AxSL}^{\mathsf{SCExt}}$ is to constrain the use of P: it isolates P from the reasoning along po that we do using $\mathsf{wp}^{\mathsf{SCExt}}$. This stratification is reflected in the model $\mathsf{wpb}^{\mathsf{SCExt}}$: the update of tied resources as performed by the view shift in FlowImp cannot affect the recursive $\mathsf{wpb}^{\mathsf{SCExt}}$ for s': we cannot use the updated resource to directly reason about s'. In Fig. 41, this is reflected by $\mathsf{wpb}^{\mathsf{SCExt}}$ being pulled out of the view shift. This is to contrast with $\mathrm{AxSL}^{\mathsf{SC}}$ (see Fig. 38), where $\mathsf{wpb}^{\mathsf{SC}}$ is on the right side of the view shift, and can therefore observe the update. This stratification is crucial for how we structure adequacy of $\mathrm{AxSL}^{\mathsf{Arm}}$, as we will show in the next section.

6.4.2 Soundness of Proof Rules. Like in $\mathrm{AxSL}^{\mathsf{SC}}$, we need to implement logical interpretation predicates and define a notion of single-step weakest precondition and Hoare triple. The soundness proofs for proof rules are then unsurprising, thanks to the substantial similarity between the models of the two wpbs, except that now more effort is needed to show that the update of τ to τ' satisfies the FlowImp predicate.

Theorem 6.2. *The* $\mathrm{AxSL}^{\mathsf{SCExt}}$ *proof rules for microinstructions are sound.*

6.5 Model and Soundness of $\mathrm{AxSL}^{\mathsf{Arm}}$

The model for $\mathrm{AxSL}^{\mathsf{SCExt}}$ is almost parametric in the memory model, in the sense that it almost works directly for $\mathrm{AxSL}^{\mathsf{Arm}}$, a logic for TinyArm with relaxed Arm-A concurrency. We elaborate the two key changes we make to adapt the $\mathrm{AxSL}^{\mathsf{SCExt}}$ model to Arm-A below. This demonstrates that the structure of the model can easily be adapted to another axiomatic memory model in which the synchronisation order (ob) and its local fragment (lob) are specified.

6.5.1 Hoare Triples. As in the other two logics, we define our Hoare triple of $\mathrm{AxSL}^{\mathsf{Arm}}$ for a microinstruction program using its weakest precondition $\mathsf{wp}^{\mathsf{Arm}}$; and define the weakest precondition using a base weakest precondition $\mathsf{wpb}^{\mathsf{Arm}}$ for an ongoing local execution state.

The model of the base weakest precondition $\mathsf{wpb}^{\mathsf{Arm}}$ has the same overall structure as in $\mathrm{AxSL}^{\mathsf{SCExt}}$, with a notable difference: because it targets the relaxed memory model of Arm-A, resources can only flow along the ob ordering. To enforce this, we make two changes in the definition of $\mathsf{wpb}^{\mathsf{Arm}}$, as depicted in Fig. 43:

- In FlowImp, we now require the quantified progress set s_{pg} to include only ob predecessors, so that one can only flow resources from nodes ordered with respect to the current event.

[6] Since τ is higher-order, the official definition actually includes a later modality on the right of the separation implication in the definition of Detach, but we have omitted that from the presentation for simplicity.

- In Detach, we require the user-provided τ_{in} to mention only lob predecessors of e, to further constrain the flow of local resources.

$$
\begin{aligned}
\mathsf{wpb}^{\mathsf{Arm}}_{tid,\Phi}\ s\ \{Q\} &\triangleq \big(s = \mathsf{Done}\ T \land \dot{\models\!\!\!\Rrightarrow} Q(T)\big) \lor \\
&\quad \left(\begin{aligned}
&s = \mathsf{Ctd}\ C \land \\
&\left(\begin{aligned}
&\forall \sigma.\ \mathsf{Valid}(\sigma.X) \mathrel{-\!\!*} (\Box\, \mathsf{SI}(\sigma)) \mathrel{-\!\!*} \forall s'.\ \langle s, \sigma\rangle \xrightarrow{tid}_{\mathsf{h}} \langle s', \sigma\rangle \mathrel{-\!\!*} \\
&\quad \forall e = \langle tid, C.T.IT.cntr\rangle, \tau.\ \big(\mathsf{IsValidEid}(e, X) * \mathsf{SI}_{\mathsf{T}}(\tau) \Rrightarrow \\
&\qquad \exists \tau'.\ \mathsf{SI}_{\mathsf{T}}(\tau') * \mathsf{FlowImp}(\sigma.X, \Phi, e, \tau, \tau') * \mathsf{wpb}^{\mathsf{Arm}}_{tid,\Phi}\ s'\ \{Q\}\big) \\
&\qquad \lor(\neg \mathsf{IsValidEid}(e, X) * \mathsf{wpb}^{\mathsf{Arm}}_{tid,\Phi}\ s'\ \{Q\})
\end{aligned}\right)
\end{aligned}\right) \\
\mathsf{FlowImp}(e, X, \Phi, \tau, \tau') &\triangleq \exists \tau_{in}, \tau_{res}, R.\ \mathsf{Detach}(X, e, \tau, \tau_{in}, \tau_{res}) * \tau' = \tau_{res}[e \mapsto R] * \\
&\qquad \forall s_{pg} \supseteq \mathsf{PredOf}(X.\mathsf{ob}, e) \land e \notin s_{pg}. \\
&\qquad \Big(\mathop{\huge *}_{(e \mapsto R_{in}) \in \tau_{in}} R_{in}\Big) * \mathsf{SIP}(\Phi, X, s_{pg}) \Rrightarrow \mathsf{SIP}(\Phi, X, s_{pg} \cup \{e\}) * R \\
\mathsf{Detach}(X, e, \tau, \tau_{in}, \tau_{res}) &\triangleq \mathsf{dom}(\tau_{in}) \subseteq \mathsf{PredOf}(X.\mathsf{lob}, e) * \forall (e \mapsto R_{in}) \in \tau_{in}. \\
&\qquad \exists R_{res} = \tau_{res}(e), R = \tau(e).\ R \mathrel{-\!\!*} (R_{in} * R_{res})
\end{aligned}
$$

Fig. 43. The model of base weakest precondition of AxSL$^{\mathsf{Arm}}$. The diff from AxSL$^{\mathsf{SCExt}}$ (highlighted in orange) reflects the shift of the synchronisation order from sc to ob.

6.5.2 Soundness of Proof Rules. With these minor modifications from AxSL$^{\mathsf{SCExt}}$, this model works for AxSL$^{\mathsf{Arm}}$, and we can prove soundness of the proof rules for microinstructions:

Theorem 6.3. *The AxSL$^{\mathsf{Arm}}$ proof rules for microinstructions are sound.*

Proof sketch. The proof is the same as the one for AxSL$^{\mathsf{SCExt}}$, except that now we flow resources along lob/ob instead of po/sc (which is possible thanks to the stratification implemented by the model, as we remarked in §6.4), and we need to deal with the dependency edges of Arm-A. □

6.5.3 Supporting Framing and Invariants. In the same way that one is used to splitting resources in separation logic, one would expect to be able to split $a \hookrightarrow (P{*}Q)$ into $(a \hookrightarrow P){*}(a \hookrightarrow Q)$. Recall that it is useful for proving examples (see Fig. 8). Modeling such splitting is, however, quite challenging due to its higher-order nature. (Interested readers may find our complex implementation using a combination of various Iris CMRAs in Appendix C.2) We address this challenge by first implementing splitting with the help of the interpretation SI_{T}: $e \hookrightarrow (P * Q) \mathrel{-\!\!*} (\forall \tau.\ \mathsf{SI}_{\mathsf{T}}(\tau) \Rrightarrow (\mathsf{SI}_{\mathsf{T}}(\tau) * e \hookrightarrow P * e \hookrightarrow Q)))$. Here, the view shift allows us to update tied assertions and the interpretation without changing the value of the global map τ. Then, the view shift structure is hidden by adapting 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 a more expressive view shift (a combination of the 'later' modality and the 'fancy update' modality) that allows opening and closing of invariants, similar to [Jung et al. 2018]. Importantly, these invariants do not enable resource transfer (which would be unsound), as they did in CSL for non-relaxed concurrency. Instead, we mainly use them to construct persistent wrappers for non-persistent resources that we want to transfer via AxSL$^{\mathsf{Arm}}$ protocols, like escrows in GPS [Kaiser et al. 2017; Turon et al. 2014].

6.5.4 Supporting Exclusives. We use the escrow pattern to support transferring non-duplicable resources at a successful read-modify-write of a given write, as outlined in §5.

Given a location x on which to use read-modify-writes to transfer an exclusive resource P, we put P into the following invariant, as part of the protocol of x:

$$\Phi(x, \_, e_w) \triangleq \boxed{P \vee \left(\exists e_r, e'_w.\, e_w\ ((\mathsf{rf}; [e_r]; \mathsf{rmw})\ \&\ \mathsf{co})\ e'_w * \mathsf{ExTok}(e'_w)\right)} * \ldots$$

A store e_w to this location merely needs to establish the invariant by sending P away (to satisfy the left disjunct). A load exclusive e_r reading from e_w can obtain the invariant, but cannot do anything more with it by itself. A po-later successful store exclusive e''_w that pairs with e_r is guaranteed to be the unique store exclusive paired with a load that reads-from the same e_w. This unicity implies that e''_w is the quantified e'_w (by graph reasoning), which allows us to open the invariant and refute the right disjunct: With the exclusive token $\mathsf{ExTok}(e'_w)$ obtained from the rule(as rmw is in ob), and the same token from the protocol, we conclude a contradiction. Therefore, a successful store exclusive makes it possible to get P from the left disjunct when opening the invariant.

We now describe the needed minor adaptions to the language semantics and the logic. (1) We add an extra bookkeeping field $\mathit{srcs}_{\mathsf{rmw}}$ (of type *option(Eid)*) to the thread state T of the opax. This new field is to remember the candidate load exclusive e that the next store exclusive may pair with, which we track with a new bookkeeping assertion $\mathsf{RmwPred}(e)$, similar to how we track $\mathit{srcs}_{\mathsf{ctrl}}$ with $\mathsf{CtrlPreds}$. (2) We extend SI_{T} with a new interpretation for the domain of the global tied-to map τ, such that it, together with $\mathsf{ExTok}(e)$, satisfy the following properties that guarantee the uniqness of a token

$$\textsc{SIT-extok-alloc}\quad \frac{e \notin \mathsf{dom}(\tau) \qquad \tau' = \tau[e \mapsto \_]}{\mathsf{SI}_{\mathsf{T}}(\tau) \Rrightarrow \mathsf{SI}_{\mathsf{T}}(\tau') * \mathsf{ExTok}(e)} \qquad\qquad \textsc{extok-excl}\quad \mathsf{ExTok}(e) * \mathsf{ExTok}(e) \vdash \bot$$

6.5.5 Pulling Out Tied Resources. In some situations, for example when considering the postcondition of a whole program, knowing exactly which event which resource comes from is not helpful. Instead, one can use a 'normal' postcondition by pulling out the tied resources from tied-to assertions (which means that we cannot reuse the specification in the proof of a larger program anymore). This only requires a minor change to the semantics of $\mathsf{wpb}^{\mathsf{Arm}}$: we replace the $\models\!\dot{\Rrightarrow} Q$ in the case handling termination with $\mathsf{PullOutTied}(\mathit{tid}, Q)$:

$$\mathsf{PullOutTied}(\mathit{tid}, Q) \approx \forall \tau.\, \mathsf{SI}_{\mathsf{T}}(\tau) \mathrel{-\!\!*} \left(\mathsf{SI}_{\mathsf{T}}(\tau) * \left(\mathop{\Huge *}_{\{R \mid (e \mapsto R) \in \tau \wedge \mathit{tid} = e.\mathit{tid}\}} R \Rrightarrow Q\right)\right)$$

$\mathsf{PullOutTied}$ requires us to establish the postcondition Q, assuming that the predicates pulled out from local tied assertions hold. Technically, the definition of $\mathsf{PullOutTied}(\mathit{tid}, Q)$ makes use of the agreement rule for a local event e, which roughly says that $\mathsf{SI}_{\mathsf{T}}(\tau) * e \hookrightarrow R$ implies that $e \mapsto R$ is in τ 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 τ map mentioned above) — we have already seen an example of how this is used, namely in the final reasoning step (in each thread) in Fig. 25.

6.5.6 Stuckness and Infinite Executions. As described in §4.3.5, we assume non-stuckness in the model of our weakest preconditions. We do not need to show that the program does not get stuck, since we only consider terminated opax traces, as we will see in the adequacy statement in the next section.

Besides, because of the open problem with infinite executions in the memory model (discussed in §4.3.6), our definitions of weakest preconditions does not take measures to handle infinite executions either.

7 ADEQUACY

The adequacy of Iris logics is usually expressed as a meta-level theorem. Generally speaking, this theorem about a program logic (in our case, Iris) extracts, from a program specifications proven in the logic, a result in the meta logic in which the program logic is implemented (in our case, this meta logic is Coq). This theorem shows that the program logic is sound, in the sense that the properties of programs proved in the logic hold in the meta logic.

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, as described in the general recipe in §6.1. The adequacy proofs of AxSL$^{\mathsf{Arm}}$ differs from — and, we argue, generalises — the general recipe (basically, the general recipe is our approach, instantiated with the same order twice, see §7.2.4 for an example) in two respects: our novel opax semantics, and the semantics model.

In this section, we first explain how to work with an opax semantics by showing the adequacy of AxSL$^{\mathsf{SC}}$. In this adequacy proof, we need to handle the opax shape, and the rely-guarantee style resource transfer. Next, we show the adequacy proof of AxSL$^{\mathsf{Arm}}$; we skip AxSL$^{\mathsf{SCExt}}$, since it has an almost identical semantic model, and thus proof, to AxSL$^{\mathsf{Arm}}$. This proof significantly differs from the previous general recipe, due to the new semantic model that enforces stratification (as described in §6.4.1). Because the stratification isolates the reasoning order (po) and the resource flowing order (ob), we have to do two separate inductions on these two separate orders.

7.1 Adequacy of AxSL$^{\mathsf{SC}}$

The statement of adequacy of AxSL$^{\mathsf{SC}}$ is as follows:

Theorem 7.1 (Adequacy of AxSL$^{\mathsf{SC}}$). *For any initial thread states $\vec{C}$ (each is a pair $\langle p, T\rangle$), meta-level propositions $\vec{P}$ (one for each thread), and valid execution graph X, we have*

$$\begin{array}{l}
\left(\text{❶} \bigwedge_{tid=1}^{n} \mathsf{Ctd}\ \vec{C}(tid) \xrightarrow{tid,X,I}{}^{*}_{\mathsf{h}}\ \mathsf{Done}\ \_\right) \Rightarrow \\
\text{❷}\exists\Phi.\vdash \left(\begin{array}{l}\text{❸}\mathsf{InitRes}(\Phi) * (\text{❹}\,\Box\,\mathsf{SI}(\langle X, I\rangle)) * \\ \mathop{\Large *}_{tid=1}^{n} \left(\text{❺}\mathsf{LSI}(\vec{C}(tid).T)_{tid} * \text{❻}\mathsf{wp}^{\mathsf{S}}_{tid,\Phi}\ \vec{C}(tid).p\ \left\{\left\lceil \vec{P}(tid)\right\rceil\right\}\right)\end{array}\right) \Rightarrow \\
\quad\left(\text{❼} \bigwedge_{tid=1}^{n} \vec{P}(tid)\right)
\end{array}$$

Here, $\vec{T}$ is a sequence of initial thread states (one for each thread). The first hypothesis ❶ ensures that the memory graph X reflects the behaviours of a complete program by assuming terminating executions of all threads starting from Ctd $\vec{C}$. The second line assumes that we have proofs in AxSL$^{\mathsf{SC}}$ of weakest preconditions (as ❻), 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 ❷), as well as agreement about the execution graph X and instruction memory I via the state interpretation (❹). In addition, we also require the allocation of initial protocol resources ❸ (for all initial writes). The weakest preconditions ❻ are for microinstructions, which takes the microinstruction program $\vec{C}(tid).p$. The post conditions are assumed to be the meta-level propositions $\vec{P}$, lifted to AxSL$^{\mathsf{SC}}$ 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$^{\mathsf{SC}}$.) Next to the weakest precondition of the thread, we require the local state interpretation ❺ for the initial thread state $\vec{C}(tid).T$. From these, adequacy tells us that ❼ $\vec{P}$ hold in the meta-logic as well.

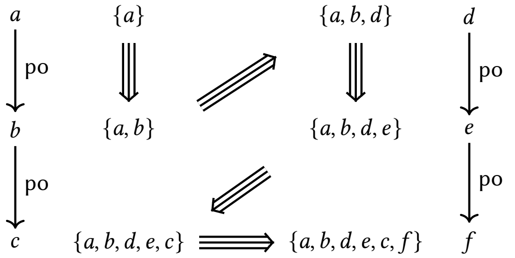


Fig. 44. An example program execution graph with two threads, above an SC memory model. Assuming sc edges from b to d, from e to c, and from c to f, $\Rrightarrow$ indicates the order we update $\mathsf{SIP}(\Phi, X, s_{pg})$, the interpretation of the progress set in the adequacy proof, where the sets of nodes indicates the advance of the progress omitting initial nodes.

The value of the adequacy theorem is that it means that we do not need to trust or even understand the intricacies of AxSL$^{\mathsf{SC}}$: once we have proved weakest preconditions for each thread using the AxSL$^{\mathsf{SC}}$ proof rules, then the adequacy theorem guarantees that the postconditions $\vec{P}$ do indeed hold at the meta-level (assuming each thread's execution terminates).

Proof Sketch. The overall proof strategy is that we first show $\models\!\dot{\Rrightarrow} \left\lceil \bigwedge_{tid=1}^{n} \vec{P}(tid) \right\rceil$, that is the goal lifted to AxSL$^{\mathsf{SC}}$, and then show that all lifted meta level propositions (under certain Iris modalities) hold in the meta logic. We only look at the first part, as the latter is exactly the soundness result of the Iris base logic.

The proof starts by allocating $\mathsf{SIP}(\Phi, X, s_{pg})$, where s_{pg} is the set of all initial (write) events, which means that we have to establish the protocol resources on all initial nodes. This is derivable from $\mathsf{InitRes}(\Phi)$ which essentially states the same thing. Next, we can unfold the model of every wp. Given the local interpretation LSI, we obtain one wpb from one wp. We then do induction on the sc order. The proof then proceeds in a way that closely follows the adequacy proof of the general recipe, as illustrated in Fig. 44. That is, we follow the sc order, which is the order that the program executes, to update logical resources using the view shift which we obtain by unfolding the model of wpb. One major difference is that, unlike SI in the general recipe, the resource we update now is SIP. Respecting the sc order, we collect events into s_{pg} via the update, and at the same time ensure that the protocol Φ is maintained by all nodes. □

Crucially, the proof would not go through if we did induction on the program execution trace, which is what the general recipe does. This is because traces of the opax semantics do not contain interleaving (they are merely thread-local traces), but the resource transformation depends on interleaving. We therefore have to do induction on a structure that contains the interleaving information, which is the sc relation of the execution graph. Technically, the $s_{pg} \supseteq \mathrm{Pred}(X.\mathrm{sc}, e)$ condition in the model of wpb enforces that one can only perform the resource update for e (making a progress) after all sc predecessors of e have been handled.

7.2 Adequacy of AxSL$^{\mathsf{SCExt}}$ and AxSL$^{\mathsf{Arm}}$

We cover the adequacy of AxSL$^{\mathsf{SCExt}}$ and AxSL$^{\mathsf{Arm}}$ together, as the similarities between their semantic models mean that their adequacy proofs also proceed similarly. We focus on the adequacy proof of AxSL$^{\mathsf{Arm}}$, and only describe its difference to the one for AxSL$^{\mathsf{SCExt}}$ at the end of this subsection.

The *statement* of adequacy for AxSL$^{\mathsf{Arm}}$ is identical to Theorem 7.1, except for that we now additionally require an SI_{T} for an empty tied-to map in the second hypothesis, meaning that, at the beginning, no resources are associated with

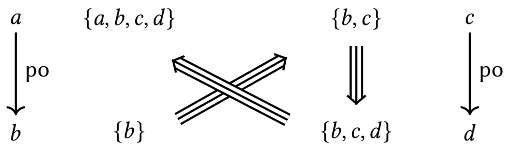


Fig. 45. An example program execution graph with two threads, above Arm-A memory model. Assuming ob edges from b to c, from c to d, and from d to a, $\Rrightarrow$ indicates the order we update $\mathsf{SIP}(\Phi, X, s_{pg})$ in the second phase of the adequacy proof, where the sets of nodes indicate the advance of s_{pg}.

any events. However, the *proof* of adequacy significantly diverges from that of Theorem 7.1, because of the stratification enforced by the semantic model of AxSL$^{\mathsf{Arm}}$.

Theorem 7.2 (Adequacy of AxSL$^{\mathsf{Arm}}$). *For any initial thread states $\vec{C}$, meta-level propositions $\vec{P}$ (one for each thread), and valid execution graph X, we have*

$$\begin{aligned}
&\left(\bigwedge\nolimits_{tid=1}^{n} \mathsf{Ctd}\ \vec{C}(tid) \xrightarrow{tid,X,I}{}^{*}_{\mathsf{h}} \mathsf{Done}\ \_\right) \Rightarrow \\
&\exists \Phi.\vdash \begin{pmatrix} \mathsf{InitRes}(\Phi) * \mathsf{SI}_{\mathsf{T}}(\emptyset) * (\Box\, \mathsf{SI}(\langle X, I\rangle)) * \\ \mathop{\ast}\nolimits_{tid=1}^{n} \left(\mathsf{LSI}(\vec{C}(tid).T) * \mathsf{wp}^{\mathsf{A}}_{tid,\Phi}\ \mathsf{Ctd}\ \vec{C}(tid).p\ \left\{\left\lceil \vec{P}(tid) \right\rceil\right\}\right)\end{pmatrix} \Rightarrow \\
&\quad \left(\bigwedge\nolimits_{tid=1}^{n} \vec{P}(tid)\right)
\end{aligned}$$

7.2.1 Overview of the Proof. Following the stratification of the semantics model, our novel adequacy proof is also stratified into two phases. We now outline the two phases informally.

Phase one, informally. First, like in some previous logics based on axiomatic models, including RSL [Vafeiadis and Narayan 2013] and GPS [Turon et al. 2014], we construct an annotated execution graph. This follows directly from the fact that the opax semantics is essentially an operational wrapper on top of an axiomatic model. Then we construct an annotation of the execution graph using flow implications from our weakest precondition. At each reduction step of the semantics, the weakest precondition remembers a flow implication for each node in the graph, and ensures that all these flow implications can be chained.

Thus, we can collect the needed flow implications at all nodes by unfolding the weakest preconditions of all threads along program order, and then connect them together to obtain a full annotation. During the annotation construction, our protocol plays a crucial role, since it specifies how resources flow across threads and is agreed upon between them, guaranteeing that the flow implications of different threads are compatible.

Phase two, informally. Second, to get an Iris-style adequacy statement in which we show all pure postconditions hold in the meta logic, we need to actually perform all the resource transformations (namely the flow implications) of the annotated graph, starting from the initial resources. This step of our proof is novel, since usually the resource transformations are simply performed *on the fly* in stronger settings, thanks to the acyclicity of po ∪ rfe in these memory models. In those settings, it is possible to update resources between program points according to the flow implications (i.e. apply them to the resources) in po ∪ rfe during the unfolding, since this order includes the po order in which weakest preconditions collect the flow implications.

On the other hand, when working with a relaxed model like that of Arm-A, the order that we rely on is ob, which does not include po, and our tied-to assertions are employed to restrict po resource flow in AxSL$^{\mathsf{Arm}}$. However, we still want syntax-oriented weakest preconditions which collect flow implications along po, as illustrated in Fig. 45. This tension poses the main challenge to the proof of adequacy, since one cannot do induction on the potentially cyclic $\mathrm{po} \cup \mathrm{ob}$. We resolve this tension by *stratifying* the usual proof procedure described in the general recipe into two phases: phase one collects flow implications along po, and phase two applies them along ob. We argue that this stratification, and the semantic model of $\mathrm{AxSL}^{\mathsf{Arm}}$ that enables it, are a generalisation of the usual Iris-based CSLs and their one-pass adequacy proofs.

In the rest of the section, we explain our novel two-phase adequacy proof in more detail, and leave the discussion on the relation to other adequacy proofs to related work (§9).

7.2.2 Phase One. The goal of phase one is to show the following lemma, from which we show the final adequacy statement in phase two.

Lemma 7.3. *For a valid execution graph X,*

$$
\begin{aligned}
&\exists \epsilon, \tau. \text{❶}\mathsf{SI}_{\mathsf{T}}(\tau) * (\text{❷}\mathsf{dom}(\tau) = \mathsf{dom}(\epsilon) = \mathsf{AllNodes}(X)) * \\
&\quad (\text{❸}\forall e \mapsto m \in \epsilon.\, \mathsf{dom}(m) \subseteq \mathsf{PredOf}(X.\mathsf{lob}, \mathrm{e})) * \\
&\quad \forall e \mapsto R \in \tau. \\
&\qquad \forall s_{pg} \supseteq \mathsf{PredOf}(X.\mathsf{ob}, e) \land e \notin s_{pg}.\, \Big(\mathsf{SIP}(\Phi, X, s_{pg}) * \text{❹} \mathop{\Large\ast}_{\_ \mapsto R_i \in \epsilon[e]} R_i\Big) \Rrightarrow \\
&\qquad\qquad\qquad\qquad \Big(\mathsf{SIP}(\Phi, X, s_{pg} \cup \{e\}) * \text{❺} \mathop{\Large\ast}_{e_o \in \mathsf{SuccOf}(X.\mathsf{lob}, \mathrm{e})} \epsilon[e_o][e] * R\Big)
\end{aligned}
$$

Generally speaking, the lemma requires a well-formed annotation on every lob edge of the execution graph. The edge annotation ϵ (of type $\mathit{Eid} \longrightarrow \mathit{Eid} \longrightarrow \mathit{iProp}$) records the history of how resources evolve and are transferred soundly along lob in a complete program execution (e.g. $\epsilon[e][e']$ is the annotation of e' lob e), as inspired by RSL/FSL. The first line of the lemma also requires a tied-to map τ, and its logical interpretation ❶$\mathsf{SI}_{\mathsf{T}}(\tau)$. This map is the final one after checking all events, which records the resources that remain on the events after flowing all the resources. To enforce that this map is indeed final, ❷ requires that the domain of τ and ϵ is all the nodes of the execution graph X. In the next line, we impose a well-formedness condition on ϵ: ❸ says that, for all mappings of e to m in ϵ, m is a node-to-resource map specifying the resources that flow to e from its lob predecessors. The last three lines relate τ and ϵ: for all dangling resources R on node e in τ, a variant of the view shift part of FlowImp holds for the protocol Φ, where ❹ is the local resources flowing into e, and ❺ is the resources flowing out from e, both along lob.

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 specified by the flow implication: we take τ_{in} as $\epsilon[e]$ for every node e. 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, which is captured as the lemma above.

Proof Sketch. In each thread, 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, by taking the union of the local ones) to obtain a global annotation ϵ, which is possible since all resource exchange across threads (as annotated on obs) are specified by the same rely-guarantee protocol Φ. □

7.2.3 Phase Two. Given the edge annotation ϵ from phase one, phase two stitches the flow implications together by induction on ob, as illustrated in Fig. 45.

Proof Sketch. 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. Finally, we combine the remaining tied-to assertions at the end of the process to show the postconditions. □

7.2.4 Adequacy of AxSL$^{\mathsf{SCExt}}$. The adequacy proof of AxSL$^{\mathsf{SCExt}}$ differs from that of AxSL$^{\mathsf{Arm}}$ only in phase two: we just need to make the induction of phase two follow sc instead of ob. This proof is effectively the proof of adequacy of AxSL$^{\mathsf{SC}}$, but divided into two phases due to the stratification imposed by the model. Recall that in the AxSL$^{\mathsf{SC}}$ proof, we do induction on the sc relation to collect flow implications and apply them on-the-fly. Here in this proof, we collect the flow implications in phase one along po and apply them in the sc order in phase two.

8 TECHNICAL REMARKS AND LIMITATIONS

8.1 Technical Improvements to the Original AxSL$^{\mathsf{Arm}}$

Although the syntax of AxSL$^{\mathsf{Arm}}$ in §5 is almost identical to the original AxSL$^{\mathsf{Arm}}$ [Hammond et al. 2024], the model presented in §6 is new, and factored cleanly around well-defined abstractions. This has two benefits: first, it makes the proof of adequacy significantly simpler, and second, it allows us to apply the same recipe to different memory models, which we illustrate using SC in AxSL$^{\mathsf{SCExt}}$.

Concretely, in the old model, we enforce the protocol on exactly the obs predecessors of the current node, which causes two major inconveniences. First, in the proof of adequacy, we have to unfold the (old) $\mathsf{FlowImp}$ predicate, and reason about the protocol resources tied to the obs predecessors of a node explicitly. For instance, we have to reason about whether an obs predecessor of a node e is also an obs predecessor of another node e' (e.g., when two loads read from the same write) in the phase two of the adequacy proof. This adds extra complexity to the proof.

Second, users of the logic can only send away persistent resources with the (old) proof rule for stores. This is because we have to require the protocol resources to be persistent, ensuring that they remained unchanged after applying the flow implication (so they can be transferred to other nodes) in the adequacy proof.

With the improved model of this paper, we use the progress set s_{pg} to track the set of events on which we have ensured that the protocol holds, and abstract the enforcement of the protocol on s_{pg} with SIP. This abstraction avoids unfolding the definition of SIP and reduces explicit resources reasoning.

8.2 Proof Effort for AxSL$^{\mathsf{Arm}}$

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 the original proof of the POPL 2024 paper after initial design work. Afterwards, the proof was simplified thanks to the improvement to the model described in §8.1, which took around a week. Writing and proving an instruction proof rule takes about half a person-day now that we have enough of them to 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 for AxSL$^{\mathsf{Arm}}$ is divided as follows:

Item	**LoC**
Prelude (incl. outcome interface and infrastructure)	~4800
Language definition and lemmas	~1100
Axiomatic model and lemmas	~3000
Iris CMRAs	~900
WPs and assertions	~3700
Proof rules and their soundness proofs	~2700
Adequacy	~1100
Examples	~3300

The purpose of this table is to give readers an impression on the scale of the mechanisation. The numbers in this tabular are not intended for direct numerical comparison with the numbers reported in the POPL 2024 paper [Hammond et al. 2024], since the mechanisation has experienced refactoring and is therefore more flexible than before.

8.3 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 §5.7). This latter order is sometimes also called *coherence*, or (to avoid confusion with the coherence *relation*, co, which is merely part of it) *extended coherence*.

The way AxSL$^{\mathsf{Arm}}$ is defined in Iris above the opax semantics using the memory model of Fig. 4 means that it captures both axioms. However, the design of the logic focuses on the external axiom, and leverages the acyclicity of ob to allow sound transfer of resources along ob. This means that AxSL$^{\mathsf{Arm}}$ cannot soundly allow transfer resources along the potentially cyclic combination of ob and extended coherence, as the phase two induction proof of the adequacy needs an acyclic order. It is always possible to reason about extended coherence by brute force in AxSL$^{\mathsf{Arm}}$, by explicit graph reasoning (as in §5.5), but this is unsatisfactory. This is a definite limitation of AxSL$^{\mathsf{Arm}}$, as many common programming and reasoning idioms rely on reasoning about coherence, in particular full GPS protocols and the notion of non-atomics.

9 RELATED WORK

There is extensive work on verification of relaxed memory models. Here we mainly discuss the line of work on separation logics starting from RSL, and only mention some other work that is closely related.

Overview. RSL [Vafeiadis and Narayan 2013], GPS [Turon et al. 2014]/GPS+ [He et al. 2016], and FSL [Doko and Vafeiadis 2016]/FSL++ [Doko 2021; Doko and Vafeiadis 2017] are defined with respect to an axiomatic memory model, namely that of C11/RC11, and their proofs of soundness are built from the ground up using non-standard models of separation logic, which (as described by Kaiser et al. [Kaiser et al. 2017, §1.2]) requires significant effort. Later logics like iGPS [Kaiser et al. 2017], Cosmos [Mével et al. 2020], and iRC11 [Dang et al. 2020], rely on (re)formulating the target relaxed memory model as an operational model to obtain an Iris-based logic with advanced features like higher-order ghost states 'for free'.

Very relaxed hardware memory models. The proofs of adequacy of RSL and FSL are somewhat similar to ours, also being 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 $\mathsf{po} \cup \mathsf{rf}$.

FSL++ [Doko 2021; Doko and Vafeiadis 2017] extends FSL with a form of ghost state (albeit one not as expressive as that of Iris), but this makes it unsound for memory models that exhibit load buffering, and so FSL++ targets RC11 [Lahav et al. 2017], a significant strengthening of C11 that requires that $\mathsf{po} \cup \mathsf{rf}$ is acyclic.

In a sense, FSL and FSL++ both allow reasoning along po, but put some guards to limit transfer of physical resources. For FSL, which has no other resources, this limits its expressivity but poses no soundness problem. For FSL++, this imposes strong requirements on the underlying memory model.

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.

Very relaxed programming language models. 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.

Less relaxed memory models. GPS [Turon et al. 2014] targets the subset of C11 featuring release stores, acquire loads, and non-atomic accesses (on which data races are undefined behaviour). GPS features ghost state (which is sound because $\mathsf{po} \cup \mathsf{rf}$ is acyclic), per-location protocols (carrying state transition system tokens), and escrows. Kaiser et al. [Kaiser et al. 2017] describe how the protocols and escrows of GPS can be defined in terms of simpler components (like invariants) in Iris, and this is part of our motivation for using Iris.

To enrich GPS with the expressive power of Iris 'for free', iGPS [Kaiser et al. 2017] is based on an operational reformulation of release-acquire. One of our contributions is to show how to reason about an axiomatic memory model directly in Iris, avoiding this operational reformulation. iRC11 [Dang et al. 2020] combines iGPS with FSL++. It targets ORC11, an operational reformulation of a fragment of RC11.

Compass [Dang et al. 2022] is a specification framework for the ORC11 memory model 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.

Cosmo [Mével et al. 2020] is a logic for the multicore OCaml memory model. The OCaml memory model is stronger and simpler than that of C11 in many respects which Cosmo leverages extensively to derive simple but powerful reasoning rules. Following iGPS, Cosmo is an instantiation of the standard Iris framework with an operational semantics, following the general recipe, and has a layered design featuring a base logic exposing memory model details and a high-level logic with almost standard CSL proof rules.

Very strong models. For other stronger models like TSO that have a simple operational model, working that close to the axiomatic model is probably more a burden, although possible. Significantly different abstractions would need to be built on top to capture this strength so that the logic is usable [Jacobs 2014; Ridge 2010; Sieczkowski et al. 2015; Wehrman 2012; Wehrman and Berdine 2011; Zhang and Feng 2014].

Per-location protocols. As the name suggests, a per-location protocol in GPS and iGPS [Kaiser et al. 2017; Turon et al. 2014] is a logical assertion dedicated to resource transfer between memory operations of a location, in contrast to invariants, which are implicitly shared among all locations. Per-location protocols make it possible to bind the physical value of a location to an abstract state of a state transition system (STS), and ensures that the evolution of the value is consistent with the transitions of the STS. This abstraction usually enables more high-level proofs (compared to their counterparts in AxSL), and can be implemented using basic Iris building blocks (invariants and ghost states), as in iGPS. The idea is based on the memory model assumption that the accesses to individual locations have SC behaviours (SC-per-location is also known coherence) which is often phrased in axiomatic memory models as an acyclicity requirement on the (extended) coherence order, eco (e.g. the Internal visibility requirement of Arm-A in Fig. 4). Intuitively, given a pre-agreed STS, the axiom prevents reading from an old state and forces to make a valid and consistent transition when writing. In the fragments of C11 that GPS and iGPS are based on, the synchronisation order $\mathsf{po} \cup \mathsf{rf}$ is included in eco, and so threads can exchange resources describing the abstract states of the same location along the eco edges by rely-guarantee reasoning.

As noted in §5.1, our AxSL protocol Φ is heavily inspired by GPS, but tailored to only support relatively simple resource transfer, due to the limited support for the coherence reasoning described in §8.3. Concretely, our logic is parametric by a concrete stateless protocol whose implementation does not rely on the coherence axiom. We leave implementing full stateful protocols in AxSL as future work, since enabling meaningful proof rules with stateful protocols requires a non-trivial extension to AxSL that solves more challenging circularity issues. Fundamentally, such an extended logic would need to support flowing resources not only along the coherence order and the synchronisation order respectively, but also between the two orders — even though the union of these two orders is potentially cyclic, as in Arm-A.

Dealing with backtracking. The trick we use to express an axiomatic memory model as an operational semantics of the shape that Iris expects is inspired by how Islaris treats its event-enriched SMT language [Sammler et al. 2022]. A $(\text{declare-const } x); s$ program can take a step to $s[x \mapsto v]$ for any value v, and an $(\text{assert } e); s$ program can take a step in one of two ways: if e evaluates to true, the program takes a step to s; and if e evaluates to false, the assert steps to the 'execution discarded' state. The definition of postcondition in Islaris then ignores discarded states.

Tracking ordering and flowing. 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.

Tracking memory events. 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.

Tied-tos and flow implications. 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.

10 CONCLUSION

The very relaxed concurrency memory models of hardware architectures like Arm-A, in which synchronisation (ob for Arm-A) does not follow program order, make syntax-directed and thread-modular reasoning challenging. The need to program directly to the hardware architecture for performance and for access to systems features makes this challenge unavoidable. Our family of logics, AxSL, addresses this challenge through assertions that track how synchronisation is induced by the program text, and makes reasoning tractable by flowing higher-order ghost state along synchronisation. This allows us to capture and thus validate key synchronisation idioms. Moreover, as demonstrated by our instantiation of AxSL to different memory models, our approach relies essentially just on the acyclicity of the synchronisation order, and should therefore apply to many 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 synchronisation but also coherence, in particular as leveraged by non-atomics, even though the union of synchronisation and coherence can have cycles.

A FURTHER PROOF RULES

A.1 MemWrite **Rule**

HT-MICRO-MEMWRITE

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

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

$$\left\{ (), D. \begin{array}{l}
\exists e.\, D = \{e\} * \text{❹}\mathsf{LastLocalWrite}(x, v) * \mathsf{PoPred}(e) * \mathsf{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\
\text{❺} \mathop{\Large\ast}_{(r \mapsto (v,E)) \in regs} r \overset{\mathrm{r}}{\mapsto} v@E * \text{❻}\mathsf{GraphFacts}'(os, vr, x, v, e, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) * \\
\text{❼} e \hookrightarrow P(x, v)
\end{array} \right\}_{tid,\Phi}$$

Fig. 46. A proof rule of AxSL$^{\mathrm{Arm}}$ for the MemWrite microinstruction. As for the MemRead rule in Fig. 22 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. 22, 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 ❶. Because a new local write is produced, whichever resource is passed in ❶, a LastLocalWrite carrying the new event is returned ❹. The definition of GraphFacts′ at ❷ and ❻ differs from the version used in Fig. 22 because write events induce a different collection of incoming edges, most importantly lacking an incoming rf edge. Similarly the definition of Flow′ ❸ 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 ❼ 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

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

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

$$\left\{ (v, D). \begin{array}{l}
\exists e, e_{w}.\, D = \{e\} * \text{❷}\mathsf{LastLocalWrite}(x, v') * \mathsf{PoPred}(e) * \mathsf{CtrlPreds}(srcs_{\mathrm{ctrl}}) * \\
\mathsf{GraphFacts}(e, os, vr, x, v, e_{w}, e_{\mathrm{po}}, srcs_{\mathrm{ctrl}}, d, regs) * \mathop{\Large\ast}_{(r \mapsto (v,E)) \in regs} r \overset{\mathrm{r}}{\mapsto} v@E * \\
\text{❸}((e \hookrightarrow P(x, v, e_{w})) \vee v = v')
\end{array} \right\}_{tid,\Phi}$$

Fig. 47. A proof rule of AxSL$^{\mathrm{Arm}}$ for the MemRead microinstruction, similar to that shown in Fig. 22, but specialised to handle the case where the current thread contains a previous local write to the address being read from.

Fig. 47 shows a variant of the MemRead rule in Fig. 22 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 ❶ instead of NoLocalWrites, which is still returned unchanged ❷. The conclusion becomes a disjunction ❸, 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.

B RECOVERING POINTS-TOS IN AXSL$^{\text{SC}}$

In this section, we discuss the connection between the graph reasoning of opax-based logics and the heap reasoning of standard CSLs. Concretely, we show formally that we can build a notion of points-tos with abstractions in AxSL$^{\text{SC}}$.

Standard CSL points-tos. We first recap the points-to assertion and two standard proof rules using it in CSL. A usual CSL points-to assertion $x \hookrightarrow v$ means that the latest value of location x is v on the shared heap. Owning this assertion grants one the exclusive right to access x with the two standard rules in Appendix B. Our goal is to obtain a definition of points-to that satisfies the same rules in AxSL$^{\text{SC}}$.

Std-ht-micro-memread

$$\{x \hookrightarrow v\}\mathsf{MemRead}\ x\ \{w.\ w = v * x \hookrightarrow v\}_{tid,\Phi}$$

Std-ht-micro-memwrite

$$\{x \hookrightarrow v\}\mathsf{MemWrite}\ x\ v'\ \left\{\_.\ x \hookrightarrow v'\right\}_{tid,\Phi}$$

Fig. 48. Standard CSL read and write rules with points-tos (if we omit the protocol Φ). We reformulate them with the microinstruction language and assume a fixed number of hardware threads.

B.0.1 *Raw Definition.*

In AxSL$^{\text{SC}}$, we model $x \hookrightarrow v$ by leveraging the fact that all observed writes on location x are ordered by co, forming a sequence of write events, and that one can only read from the latest write (head) of the sequence with value v: reading from outdated writes would violate the acyclicity requirement. We define a raw points-to to capture these observations:

$$x \hookrightarrow_{raw} v@e_{lst} \triangleq \exists \gamma_x.\, \boxed{\circ\,[x \mapsto \gamma_x]}^{\gamma} * \exists ch.\, \boxed{\bullet\,(e_{lst} :: ch)}^{\gamma_x} * \mathsf{IsLastWrite}(e_{lst}, ch) * e_{lst}{:}\mathsf{W}\ x\ v$$

This raw definition asserts that the globally co-latest write of x has event ID e_{lst} and value v. We use two ghost states in the definition to capture this. $\boxed{\circ\,[x \mapsto \gamma_x]}^{\gamma}$ is a persistent fact binding location x to an unique ghost name γ_x. γ_x is the ghost name of a list of event IDs $\boxed{\bullet\,(e_{lst} :: ch)}^{\gamma_x}$ which models the sequence of writes of x (the head is the lastest). This ghost list ensures that updates of the sequence of the writes are monotone (list only grows), and one can at anytime take (persistent) snapshots of the list: $\boxed{\circ\, ch'}^{\gamma_x}$ such that ch' is a sub-list. The IsLastWrite predicate establishes that e_{lst} is indeed the co-latest write in the list with graph facts, and finally, $e_{lst}{:}\mathsf{W}\ x\ v$ remembers the value v.

Leveraging protocol. To ensure that the sequence of writes is well-synchronised with the program execution, that is, all writes one has reasoned about were added to the sequence, and, dually, reads can only read from the writes in the sequence, we use our rely-guarantee protocol Φ. As a first step, we fix the protocol using a new predicate WriteOf(x, e) as $\Phi(x, v, e) \triangleq \mathsf{WriteOf}(x, e)$ (this can be further generalised, but we keep it simple for now):

$$\mathsf{WriteOf}(x, e) \triangleq \exists \gamma_x.\, \boxed{\circ\,[x \mapsto \gamma_x]}^{\gamma} * \exists ch.\, \boxed{\circ\,(e :: ch)}^{\gamma_x} * e{:}\mathsf{W}\ x\ v$$

The new predicate asserts that e is an observed write event of x, and thus is included in the write sequence, depicted as SC-PT-RAW-AG. As we will see below, this is crucial for concluding that only the latest write is readable, mimicking the semantics of points-tos.

$$\textsc{SC-pt-raw-ag}\quad \frac{x \hookrightarrow_{raw} v@e * \mathsf{WriteOf}(x, e')}{e' \ \mathsf{co}^{*} \ e} \qquad \textsc{SC-pt-raw-sht}\quad x \hookrightarrow_{raw} v@e \Rrightarrow \mathsf{WriteOf}(x, e) \qquad \textsc{SC-pt-raw-upd}\quad \frac{e \ \mathsf{co} \ e' * e' {:} \mathsf{W} \ x \ v'}{x \hookrightarrow_{raw} v@e \Rrightarrow x \hookrightarrow_{raw} v'@e'}$$

$$\textsc{SC-pt-wo-pers}\quad \frac{\mathsf{WriteOf}(x, e)}{\mathsf{WriteOf}(x, e) * \mathsf{WriteOf}(x, e)}$$

Fig. 49. Selected operations of the raw points-to and WriteOf

B.0.2 Full Definition. The full points-to assertion is modeled as a monotone predicate over the po-latest event e of a thread, using the raw definition:

$$\llbracket x \hookrightarrow v \rrbracket \triangleq \lambda e. \exists e_{lst}. x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e$$

where the notation $e \leq_{\mathcal{R}} e'$ means that either the two events are identical, or e is $\mathcal{R}$-ordered before e' (in contrast, $\geq$ means identical or after). We instantiate this relation to sc, requiring that the latest write e_{lst} happens before e. We also need to monotonise all other assertions of the base AxSL$^{\mathrm{SC}}$ logic. Most of them are standard, for instance below is how we monotonise the weakest precondition.

$$\llbracket \{P\} i \ \{w. Q(w)\}_{tid,\Phi} \rrbracket \triangleq$$
$$\lambda e. \forall e' \geq_{\mathsf{po}} e. \left\{\llbracket P \rrbracket(e') * \mathsf{PoPred}(e')\right\} i \ \left\{v. \exists e'' \geq_{\mathsf{po}} e'. \llbracket Q(v) \rrbracket(e'') * \mathsf{PoPred}(e'')\right\}_{tid,\Phi}$$

B.0.3 Proving Standard CSL Rules. We now sketch the soundness proof of the two classic CSL rules in AxSL$^{\mathrm{SC}}$. The proof starts with unfolding all the monotone predicates then proceeds using AxSL$^{\mathrm{SC}}$ proof rules.

Store. For STD-HT-MICRO-MEMWRITE, after unfolding, we need to show:

$$\left\{ \ \mathsf{PoPred}(e) * \exists e_{lst}. x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e \ \right\}$$
$$\mathsf{MemWrite} \ x \ v'$$
$$\left\{(). \exists e' \geq_{\mathsf{po}} e. \quad \mathsf{PoPred}(e') * \exists e_{lst}. x \hookrightarrow_{raw} v'@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e' \ \right\}_{tid,\Phi}$$

We proceed with the base rule SC-HT-MICRO-MEMWRITE, picking e as e_{po}. We have the following view shift as a sub-goal, which guarantees that the protocol is enforced on the new write (namely the new write is added to the sequence as the latest):

$$\forall e_w. \left(\mathsf{GraphFactsW}(e_w, x, v', e) * \exists e_{lst}. x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e\right)$$
$$\Rrightarrow \ (\mathsf{WriteOf}(x, e_w)) * \exists e_{lst}. x \hookrightarrow_{raw} v'@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e_w$$

We update the raw points-to assertion to $x \hookrightarrow_{raw} v'@e_w$ by SC-PT-RAW-UPD and then take a snapshot with SC-PT-RAW-SHT. The remaining goal $e_w \leq_{\mathsf{sc}} e_w$ is trivial. Finally, we conclude the proof with the rule of consequence.

Load. In the case of Std-ht-micro-memread, we need to show:

$$\left\{ \mathsf{PoPred}(e) * \exists e_{lst}.\, x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathrm{sc}} e \right\}$$
$$\mathsf{MemRead}\ x$$
$$\left\{ w.\, \exists e' \geq_{\mathrm{po}} e.\quad w = v * \mathsf{PoPred}(e') * \exists e_{lst}.\, x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathrm{sc}} e' \right\}_{tid,\Phi}$$

We proceed by applying SC-ht-micro-memread. We need to prove the following view shift which captures that we are reading from a write that is in the sequence:

$$\forall e_r, w, e_w.\, (\mathsf{GraphFactsR}(e_r, x, v, e_w, e) * \exists e_{lst}.\, x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathrm{sc}} e * \mathsf{WriteOf}(x, e_w))$$
$$\Rrightarrow\ w = v * \exists e_{lst}.\, x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathrm{sc}} e_r * \mathsf{WriteOf}(x, e_w)$$

By SC-pt-raw-ag, we know $e_w \leq_{\mathrm{co}} e_{lst}$, namely the write e_w that we are reading from is one of the observed writes. We show $w = v$ by showing $e_w = e_{lst}$, that is, we can only read from the latest write. This is done by showing a violation of the acyclicity of sc in the other case when reading from an old write (e_w co e_{lst}). The problematic cycle is e_{lst} sc e po e_r fr e_{lst} where e_r fr e_{lst} is induced from e_w rf e_r and e_w co e_{lst}. Again, we conclude the proof with the rule of consequence.

C AN IMPLEMENTATION OF LOGICAL INTERPRETATIONS AND ASSERTIONS FOR AXSL$^{\mathrm{Arm}}$

This section requires knowledge on the built-in CMRA constructors of Iris [7].

C.1 Reserved Ghost Names

We have a collection of reserved ghost names that assert ownership of various ghost resources, which are used to implement state interpretations and assertions of AxSL$^{\mathrm{Arm}}$. We list their notations and usages in Fig. 50.

Notation	Global	Notation	Local
γ_G	Graph facts	γ_R	Register points-tos
γ_I	Instruction points-tos	γ_L	Local writes
γ_{T}	Tied-tos	γ_P	po predecessors
γ_E	Exclusive tokens	γ_C	ctrl predecessors
		γ_M	rmw predecessors

Fig. 50. Each thread has its own set of local ghost names.

C.2 Global Assertions, SI, and SI$_{\mathrm{T}}$

Persistent facts. The *Ag* constructor of Iris is used to implement graph and instruction memory. The ghost resources and state interpretation SI using it are therefore persistent.

[7] There is not a comprehensive documentation for all the constructors, we therefore refer readers to the Iris Coq repository for details: https://gitlab.mpi-sws.org/iris/iris/-/tree/master/iris/algebra.

$$\mathsf{SI}(\sigma) \triangleq \boxed{\mathrm{ag}(\sigma.X) : \mathit{Ag}(\mathit{Graph})}^{\gamma_G} * \boxed{\mathrm{ag}(\sigma.I) : \mathit{Ag}(\mathit{InstMem})}^{\gamma_I}$$
$$a \;\mathsf{R}\; b \triangleq \exists X. \boxed{\mathrm{ag}(X) : \mathit{Ag}(\mathit{Graph})}^{\gamma_G} * (a, b) \in X.\mathsf{R} \quad \text{(R is a relation)}$$
$$a{:}\mathsf{E} \triangleq \exists X. \boxed{\mathrm{ag}(X) : \mathit{Ag}(\mathit{Graph})}^{\gamma_G} * X(a) = \mathsf{E} \quad \text{(E is an event)}$$
$$a \overset{\mathrm{i}}{\mapsto} i \triangleq \exists I. \boxed{\mathrm{ag}(I) : \mathit{Ag}(\mathit{InstMem})}^{\gamma_I} * I(a) = i$$

Tied-to. We interpret the tied-to map τ (of type $\mathit{Eid} \longrightarrow \mathit{iProp}$) with two levels of indirection. First, instead of constructing a complex CMRA that directly takes the map, we use a simpler CMRA *GhostMapAg* for a map *em* that only goes to *GName* from *Eid*, where *GName* is used to manage more ghost states using simpler CMRAs. *GhostMapAg* is a standard authoritative ghost map CMRA constructor except for being insert-only (the fragmental views are persistent). Second, for every $e : \mathit{Eid}$, the corresponding ghost name $\gamma = \mathit{em}(e)$ is used to track another map γm from *GName* to *iProp*. This map allows us to track fractions of the proposition that tied to e, which is crucial for splitting tied-to assertions. We use *SetDisjAuth* and *SavedProp* of Iris for this map. *SetDisjAuth* is an authoritative ghost set (with disjoint union) CMRA constructor that we use to track a collection of ghost names, each of which owns a fraction of the tied proposition of a node using the *SavedProp* CMRA.

$$\begin{aligned}\mathsf{SI}_{\mathsf{T}}(\tau) \triangleq\ & \exists \mathit{em} : \mathit{Eid} \longrightarrow \mathit{GName}. \boxed{\bullet \mathit{em} : \mathit{GhostMapAg}(\mathit{Eid}, \mathit{GName})}^{\gamma_{\mathsf{T}}} * \mathrm{dom}(\mathit{em}) = \tau * \\ & \mathop{\Huge *}_{(e \mapsto \gamma) \in \mathit{em}} \Big(\exists \gamma m : \mathit{GName} \longrightarrow \mathit{iProp}. \boxed{\bullet \mathrm{dom}(\gamma m) : \mathit{SetDisjAuth}(\mathit{Eid})}^{\gamma} * \\ & \mathop{\Huge *}_{(\gamma' \mapsto R) \in \gamma m} \boxed{\mathrm{sp}(1/2, R) : \mathit{SavedProp}}^{\gamma'} * \triangleright \square (\tau(e) \twoheadrightarrow \mathop{\Huge *}_{(\_ \mapsto R) \in \gamma m} R)\Big)\end{aligned}$$
$$\begin{aligned}a \hookrightarrow P \triangleq\ & \exists \gamma, \gamma'. \boxed{\circ e \mapsto \gamma : \mathit{GhostMapAg}(\mathit{Eid}, \mathit{GName})}^{\gamma_{\mathsf{T}}} * \boxed{\circ\{\gamma'\} : \mathit{SetDisjAuth}(\mathit{Eid})}^{\gamma} * \\ & \boxed{\mathrm{sp}(1/2, P) : \mathit{SavedProp}}^{\gamma'}\end{aligned}$$

Token for exclusives. We extend SI_{T} with $\boxed{\bullet(\mathrm{dom}(\tau)) : \mathit{SetDisjAuth}}^{\gamma_E}$, and define $\mathsf{ExTok}(e)$ as $\boxed{\circ\{e\} : \mathit{SetDisjAuth}}^{\gamma_E}$.

C.3 Local Assertions and LSI

Register points-to. We interpret the register file and define a register points-to using the standard authoritative ghost map CMRA constructor of Iris, *GhostMap*.

$$r \overset{\mathrm{r}}{\mapsto} v@E \triangleq \boxed{\circ r \mapsto (v, E) : \mathit{GhostMap}(\mathit{RegName}, \mathit{Val} \times \mathit{Set(Eid)})}^{\gamma_R}$$

Local writes. We implement the local write assertions using the same ghost map CMRA, but on a map lm from *Addr* to *option(Eid)*, which we associate with the execution graph in LW in LSI. LW contains the authoritative view of the map, requiring that for any location a if $a \mapsto \mathsf{Some}(e) \in lm$ then e is indeed the latest write of a that the thread has been observed at the current program point; and if $a \mapsto \mathsf{None} \in lm$, no writes of a have been observed.

$$\begin{aligned}\mathsf{LW}(\mathit{cntr})_{\mathit{tid}} \triangleq\ & \exists X. \boxed{\mathrm{ag}(X) : \mathit{Ag}(\mathit{Graph})}^{\gamma_G} * \exists lm : \mathit{Addr} \longrightarrow \mathit{option(Eid)}. \\ & \boxed{\bullet lm : \mathit{GhostMap}(\mathit{Addr}, \mathit{option(Eid)})}^{\gamma_L} * \\ & (\forall a \mapsto \mathsf{Some}(e) \in lm.\ \mathsf{IsLastWriteOn}(a, e, \mathsf{OldWrites}(X, \mathit{cntr})_{\mathit{tid}})) * \\ & (\forall a \mapsto \mathsf{None} \in lm.\ \mathsf{NoWritesOn}(a, \mathsf{OldWrites}(X, \mathit{cntr})_{\mathit{tid}}))\end{aligned}$$
$$\mathsf{OldWrites}(X, \mathit{cntr})_{\mathit{tid}} \triangleq \mathsf{filter}\ (\lambda(e, E).\ e.\mathit{tid} = \mathit{tid} \wedge e.\mathit{cntr} < \mathit{cntr} \wedge \mathsf{IsWrite}(E))\ X.\mathsf{E}$$

We define the assertions as just fragmental views of the map.

$$\begin{aligned}\mathsf{NoLocalWrites}(a) &\triangleq \boxed{\circ a \mapsto \mathsf{None} : \mathit{GhostMap}(\mathit{Addr}, \mathit{option(Eid)})}^{\gamma_L}\\ \mathsf{LastLocalWrite}(a, e) &\triangleq \boxed{\circ a \mapsto \mathsf{Some}(e) : \mathit{GhostMap}(\mathit{Addr}, \mathit{option(Eid)})}^{\gamma_L}\end{aligned}$$

po *predecessors.* We define a new CMRA *MonoNatPair* for implementing PoPred which tracks the latest po predecessor. This CMRA is inspired by the *MonoNat* CMRA of Iris, which ensures that a natural number is increment-only. We take the idea, and modify it to take a pair of natural numbers, and change the order to the lexicographical order. We use the CMRA to track a lower bound of $IT.cntr$, the counter used to generate fresh *Eid*, in an indirect way in the PoP predicate in LSI. We have a tweaked oneshot CMRA $\boxed{o}^{\gamma_P}$ where o can be a *splittable* pending or $\mathsf{shot}(\gamma)$. In the latter case, γ owns a half of the authoritative view of *cntr*: $\boxed{\bullet^{1/2}\, cntr : \mathit{MonoNatPair}}^{\gamma}$. We use the other half to implement the PoPred assertion. The assertion has a persistent variant PoPred′ which we implement using the fragmental view, which can also be used to form po edges.

$$\begin{aligned}\mathsf{PoPred}(a) &\triangleq \exists\gamma.\, \boxed{\mathsf{shot}(\gamma)}^{\gamma_P} * \boxed{\bullet^{1/2}\, a.cntr : \mathit{MonoNatPair}}^{\gamma}\\ \mathsf{PoPred}(-) &\triangleq \boxed{{}^{1/2}\, \mathsf{pending}}^{\gamma_P}\\ \mathsf{PoPred}'(a) &\triangleq \exists\gamma.\, \boxed{\mathsf{shot}(\gamma)}^{\gamma_P} * \boxed{\circ a.cntr : \mathit{MonoNatPair}}^{\gamma}\end{aligned}$$

$$\begin{aligned}\mathsf{PoP}(cntr)_{tid} \triangleq \exists X.\, &\boxed{\mathsf{ag}(X) : \mathit{Ag}(\mathit{Graph})}^{\gamma_G} * \boxed{{}^{1/2}\, \mathsf{pending}}^{\gamma_P} \lor \big(\exists\gamma, cntr' < cntr.\\ &\boxed{\mathsf{shot}(\gamma)}^{\gamma_P} * \boxed{\bullet^{1/2}\, cntr' : \mathit{MonoNatPair}}^{\gamma} * \mathsf{IsValidEid}(\langle tid, cntr' \rangle, X)\big)\end{aligned}$$

ctrl *predecessors.* We use $\mathit{DfracAgree}(\mathit{Set(Eid)})$ to implement CtrlPreds. CtrlPreds contains half of the CMRA, and CoP contains the other half to track $T.srcs_{\mathsf{ctrl}}$.

$$\begin{aligned}\mathsf{CtrlPreds}(s) &\triangleq \exists s' \supseteq s.\, \boxed{{}^{1/2}\, s' : \mathit{DfracAgree}(\mathit{Set(Eid)})}^{\gamma_C}\\ \mathsf{CoP}(srcs_{\mathsf{ctrl}}) &\triangleq \boxed{{}^{1/2}\, srcs_{\mathsf{ctrl}} : \mathit{DfracAgree}(\mathit{Set(Eid)})}^{\gamma_C}\end{aligned}$$

rmw *predecessors.* We use $\mathit{DfracAgree}(\mathit{option(Eid)})$ to implement RmwPred. Similar to how CtrlPreds is implemented, we divide the CMRA into two halves, put one half in LSI, and use the other as the assertion.

$$\begin{aligned}\mathsf{RmwPred}(e) &\triangleq \boxed{{}^{1/2}\, \mathsf{Some}(e) : \mathit{DfracAgree}(\mathit{option(Eid)})}^{\gamma_M}\\ \mathsf{RmwPred}(-) &\triangleq \boxed{{}^{1/2}\, \mathsf{None} : \mathit{DfracAgree}(\mathit{option(Eid)})}^{\gamma_M}\\ \mathsf{RoP}(srcs_{\mathsf{rmw}}) &\triangleq \boxed{{}^{1/2}\, srcs_{\mathsf{rmw}} : \mathit{DfracAgree}(\mathit{option(Eid)})}^{\gamma_M}\end{aligned}$$

mrd. We omit *mrd* in the presentation of the proof rules. We can have a ghost state using $\mathit{DfracAgree}(\mathit{Set(Eid)})$ to track the set of instruction internal memory reads, or we can just expose it to the weakest preconditions for microinstructions, and later hide it in the proof rules for instructions (since it is always empty at the beginning of an instruction).

Local state interpretation. Finally, we define LSI:

$$\begin{aligned}\mathsf{LSI}(T)_{tid} \triangleq\ &\boxed{\bullet T.regs : \mathit{GhostMap}(\mathit{RegName}, \mathit{Val} \times \mathit{Set(Eid)})}^{\gamma_R} * \mathsf{PoP}(T.IT.cntr)_{tid} *\\ &\mathsf{CoP}(T.srcs_{\mathsf{ctrl}}) * \mathsf{RoP}(T.srcs_{\mathsf{rmw}}) * \mathsf{LW}(T.IT.cntr)_{tid}\end{aligned}$$

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. 2024. An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic. *Proc. ACM Program. Lang.* 8, POPL, Article 21 (jan 2024), 34 pages. https://doi.org/10.1145/3632863

Mengda He, Viktor Vafeiadis, Shengchao Qin, and João F. Ferreira. 2016. Reasoning about Fences and Relaxed Atomics. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*. IEEE Computer Society, 520–527. https://doi.org/10.1109/PDP.2016.103

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

Bart Jacobs. 2014. *Verifying TSO Programs (Report CW660)*. Technical Report.

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

Jonas B. Jensen, Nick Benton, and Andrew Kennedy. 2013. High-Level Separation Logic for Low-Level Code. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 301–314. https://doi.org/10.1145/2429069.2429105

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.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: a concurrent separation logic for multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 96 (aug 2020), 29 pages. https://doi.org/10.1145/3408978

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

Tom Ridge. 2010. A Rely-Guarantee Proof System for x86-TSO. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6217)*, Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani (Eds.). Springer, 55–70. https://doi.org/10.1007/978-3-642-15057-9_4

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

Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. 2015. A Separation Logic for Fictional Sequential Consistency. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 736–761. https://doi.org/10.1007/978-3-662-46669-8_30

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

Ian Wehrman. 2012. *Weak-Memory Local Reasoning (Dissertation draft).* Ph. D. Dissertation. University of Texas at Austin.

Ian Wehrman and Josh Berdine. 2011. A proposal for weak-memory local reasoning. In *Low-level languages and applications (LOLA).*

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

Yang Zhang and Xinyu Feng. 2014. *Program Logic for Local Reasoning in TSO.* Technical Report.