Separation Logic for Low-level and Realistic Programs

Zongyuan Liu

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Separation Logic for Low-level and Realistic Programs

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Zongyuan Liu
March 1, 2025

Abstract

Separation logic is a powerful formal verification technique for ensuring the correctness and security of software. However, applying separation logic to low-level programs presents a key challenge: capturing the complexities of realistic system modelling while maintaining modularity for compositional reasoning.

This dissertation develops novel separation logics to address two challenging aspects of low-level systems: (1) virtual memory and memory sharing hypercalls in system software and (2) concurrency under relaxed hardware architectures. VMSL is a separation logic for reasoning about virtual machines interacting through hypercalls in a virtualized environment. It enables VM-local reasoning and captures the desired memory isolation property using a logical relation. AxSL is a concurrent separation logic framework for reasoning about relaxed memory concurrency, instantiated for the highly relaxed Arm-A architecture. It ensures sound and thread-modular verification even in the presence of complex out-of-order executions. AxSL+ extends AxSL with mixed-order reasoning to enhance verification tractability. It supports constructing high-level abstractions based on multiple consistency orders of a memory model.

These contributions establish separation logic as a modular and foundational approach to verifying low-level software with dynamic memory access control and relaxed concurrency, laying the groundwork for reliable verification of real-world software.

Resumé

Separationslogik er en kraftfuld formel verifikationsteknik, der kan bruges til at sikre korrekthed og sikkerhed af software. Indtil nu er separationslogik mestendels blevet udforsket for modeller af programmeringssprog som abstraherer fra væsentlige aspekter af realistiske lavniveau-programmer og -systemer som er meget komplekse og dermed meget udfordrende at modellere og ræsonnere om.

Denne afhandling udvikler nye separationslogikker til at løse to udfordrende aspekter af lavniveau-systemer: (1) virtuel hukommelse og såkaldte "hypercalls" i systemsoftware og (2) flertrådede programmer i "relaxed memory" hardwarearkitekturer. VMSL er en separationslogik til at ræsonnere om virtuelle maskiner, der interagerer gennem "hypercalls" i et virtualiseret miljø. Det tillader VM-lokale ræsonnementer, og beskriver den ønskede isoleringsegenskab for hukommelse ved brug af en logisk relation. AxSL er et separationslogikframework til at ræsonnere om flertrådet "relaxed" hukommelse, instantieret for den meget "relaxed" Arm-A arkitektur. Det sikrer sund og trådmodulær verifikation, selv i tilstedeværelsen af komplekse udførelser af programmer. AxSL+ udvider AxSL med "mixed-order" ræsonnementer for at lette beviser i logikken. AxSL+ understøtter konstruktionen af højniveausabstraktioner baseret på flere konsistente ordninger af hukommelsesmodeller.

Disse bidrag etablerer separationslogik som en grundlæggende og modulær tilgang til verifikationen af lavniveau-programmer med dynamisk kontrol af hukommelsesbrug og "relaxed" parallelitet, og derved skaber afhandlingen grundlaget for pålidelig verifikation af lavniveau-system software i den virkelige verden.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Lars Birkedal, for his guidance, support, and continuous encouragement throughout my Ph.D. His mentorship in my academic growth has been truly inspiring and instrumental, and I am deeply grateful for his efforts in always ensuring the best possible path for me. Lars is a true role model for me.

I sincerely appreciate my co-advisor, Jean Pichon-Pharabod, for his generous support, for the countless technical discussions and chats on various topics, and for introducing me to relaxed memory when I was searching for a new research direction. I feel truly fortunate to have had Lars and Jean as my Ph.D advisors. Thank you for making my Ph.D journey such a rewarding experience.

I extend my appreciation to my co-authors for the insightful discussions during our collaborations, which have resulted in successful work; to my friends and LogSem colleagues for all the interesting and humorous kitchen chats, outdoor adventures, and bouldering sessions; and to my parents for their constant support and care, despite the physical distance between us.

Last but not least, I would like to give a special thanks to Meng. Your companionship, through both the good and bad days of the past three and a half years, has meant a great deal to me. I am fortunate to have you by my side.

Zongyuan Liu,
Aarhus, 1st March 2025.

Contents

Abstract i

Resumé iii

Acknowledgments v

Contents vii

I Overview **1**

1 Introduction 3
- 1.1 Formal Verification of Programs . . . 3
- 1.2 Scope and Thesis . . . 4

2 Background 7
- 2.1 Separation Logic . . . 8
- 2.2 Iris: A Higher-Order Concurrent Separation Logic Framework . . . 14
- 2.3 Relaxed Memory Concurrency . . . 26
- 2.4 Further Related Work . . . 36

3 Results 41
- 3.1 Coq/Rocq Mechanisations . . . 41
- 3.2 Personal Contributions . . . 42
- 3.3 Structure . . . 42

II Publications **45**

4 VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A 47

4.1 Introduction . . . 48
4.2 Formalising A Substantial Fragment of the HVC ABIs . . . 51
4.3 Reasoning about Communicating VMs . . . 56
4.4 Reasoning in the Presence of Unknown VMs . . . 65
4.5 Related Work . . . 73
4.6 Conclusion . . . 76

5 An Axiomatic Basis for Computer Programming on Relaxed Hardware Architectures: The AxSL Logics 77
5.1 Introduction . . . 78
5.2 Context: Program Logics and Relaxed Concurrency . . . 82
5.3 Key Ideas . . . 84
5.4 The Languages . . . 90
5.5 The Logics . . . 99
5.6 Model and Soundness . . . 118
5.7 Adequacy . . . 132
5.8 Technical Remarks and Limitations . . . 138
5.9 Related Work . . . 143
5.10 Conclusion . . . 146

6 First Steps towards AxSL+ 149
6.1 Introduction . . . 149
6.2 Context . . . 150
6.3 Technical Overview . . . 153
6.4 An AxSL+ Instance for Non-Atomics: AxSL+^{NA} . . . 160
6.5 Semantic Model . . . 166
6.6 Adequacy . . . 174
6.7 Further Steps . . . 178
6.8 Conclusion . . . 183

Bibliography 185

Part I

Overview

1

Introduction

1.1 Formal Verification of Programs

Formal verification of programs is the process of formally *proving* that a program does not exhibit unexpected behaviours with respect to its *specification*. This methodology complements program testing, as each approach has distinct advantages: testing validates a subset of executions at runtime, providing empirical confidence, while formal verification ensures correctness across *all* possible executions through static reasoning, offering rigorous guarantees.

To conduct verification formally in a verification system, one must first define *models* and *specifications*:

Models A mathematical model is required to describe the possible behaviours of the program. This can be, for example, formal semantics of the programming language in which the program is written, which define the *meaning* of the program. Real-world systems, however, are often highly complex, making it infeasible to capture every detail precisely. Thus, the model is usually only an approximation and/or an abstraction of the realistic behaviours. Another common compromise is simplifying the model to omit details irrelevant to the properties being verified. Such simplifications can significantly reduce verification complexity but may introduce unsoundness and are sometimes unavoidable due to constraints in verification systems (*e.g.*,to guarantee termination of automated systems).

Specifications The expected properties or sets of desired behaviours of a program must be defined with respect to the model. These specifications can be expressed directly in the model's language or in that of the verification system, which should relate to the model. A fundamental property of interest is *functional correctness* which ensures the correctness of the return value given a specific input and initial program state. Security properties are another category of interest, such as ensuring that a program does not leak confidential information and is robust against malicious attacks.

Approaches For a given verification problem, a verification system typically approaches it in one of the following two ways:

Enumeration This algorithmic approach systematically enumerates all possible program outcomes according to the model and verifies their correctness individually. State-space explosion often occurs when the number of possible states in the model grows exponentially, making it computationally infeasible to verify large or complex systems using this approach. Since enumeration techniques rely on exhaustively exploring all possible states, they are not well-suited for verifying infinite-state systems where the number of states is unbounded. A notable example of enumeration-based verification is model checking, which systematically explores all reachable states of a system, often leveraging various techniques to mitigate state-space explosion, *e.g.*,only exploring an abstract state-space of the system.

Deduction This proof-based approach translates the verification problem into logical formulas that quantify over all possible program executions. The correctness of these formulas is then established using a proof system with axioms and inference rules. The proof search can be performed almost automatically with minimal human intervention, requiring only necessary hints, or interactively. The former is efficient but limited to verifying relatively weak properties, while the latter provides greater flexibility, allowing reasoning about more expressive properties at the cost of significant manual effort. A key example of the deductive approach is program logic, which includes formalisms such as Hoare logic to reason about program correctness. The topic of this thesis, separation logic, is an extension of Hoare logic.

1.2 Scope and Thesis

This dissertation aims to advance the theory of *foundational* and *modular* verification of *low-level and realistic* programs using *separation logic*.

Foundational Verification tools (particularly automated ones) often have a complicated logic baked-in to reason about intricate program behaviours, which leads to a large trusted computing base. This reduces the assurance of verification results, as a single error in the logic can compromise the entire outcome. One approach to making formal verification more trustworthy is to make it *foundational*: foundational verification [Appel, 2001] enhances reliability by producing proofs that can be checked by *proof assistants*. The trusted computing base of various mainstream proof assistants (*e.g.*, HOL, Coq/Rocq, Lean) is limited to their relatively simple yet expressive kernels, which implement well-established core calculi. By specifying program behaviours as theorems and proving them in proof assistants, one can attain higher confidence in the verification results.

Foundational verification has been successfully applied at scale to practical

and realistic software systems. Among them, Compcert [Leroy, 2009] is a formally verified C compiler that compiles a large subset of C. It is a practical compiler that implements a collection of optimisation passes, both developed and formally verified in the Coq/Rocq proof assistant. seL4 [Klein et al., 2010, 2009] is a formally verified operating system (OS) kernel. This high-performance OS kernel provides strong security guarantees, such as isolation for applications running on top, and is verified using the Isabelle/HOL theorem prover.

Modular The exact notion of modularity varies depending on the context, but generally means *decomposing* a large and complicated verification task into smaller, more manageable ones. This idea makes program verification scalable. For instance, CompCert's compiler correctness is verified modularly: its correctness result, stating that the compilation preserves the semantics of source programs, is composed of the correctness of all compilation passes. For concurrent programs, *thread-local reasoning* [O'Hearn, 2007] enables verifying each thread *in isolation*, disregarding non-interfering behaviours of other threads. This is a key form of modularity that underpins many effective verification techniques. Another crucial aspect is specification modularity, which enables proof reuse: once a module is verified against a strong modular specification, subsequent client verification can rely on this specification without the need to reverify the module itself.

Separation logic Separation logic has proven to be a powerful deductive verification technique that supports modularity, which we elaborate on in Section 2.1. This dissertation explores the theoretical foundations of separation logic, which is particularly significant because separation logic is not only a widely used technique in foundational verification, but has also been adapted into various automated tools. These include program verifiers [Jacobs et al., 2011; Müller et al., 2016; Pulte et al., 2023], static analysers [Calcagno and Distefano, 2011; Le et al., 2022], testers [Banerjee et al., 2025; Nguyen et al., 2008], and synthesisers [Polikarpova and Sergey, 2019], among others. These tools integrate the principles of separation logic and facilitate the practical verification of substantial codebases. The results of this dissertation are thus intended to lay the foundation for large scale program verification.

Low-level and realistic Separation logic has been successfully applied to reasoning about programs modularily against relatively idealised language semantics. However, as formal verification becomes more widespread and its theories and tools become more mature, it is increasingly feasible and desirable to base verification on more *realistic modelling*, which ensures better alignment with real-world software and hardware behaviours. This dissertation focuses on *low-level* computer system behaviours that are often abstracted away from high-level languages.

This dissertation investigates the following question:

Can the modularity benefits of separation logic be preserved for low-level languages with more realistic modelling?

with two aspects of realistic modelling:

Virtual memory and hypercalls System software employs virtual memory to regulate memory access by low-privileged units. It does so by controlling the translation from virtual addresses to physical addresses. A hypervisor, for instance, manages the memory access of its clients — virtual machines (VMs) — by maintaining page tables that define these mappings, which the hardware consults to perform address translation. One of the primary objectives of this access control mechanism is to enforce *memory isolation*, ensuring that each VM's memory remains private unless explicitly shared with other VMs.

A key question is whether hypervisors can still guarantee the intended isolation when VMs invoke hypercalls for sharing their private memory. This dissertation studies the official FF-A hypercalls in the Arm architecture by modelling their behaviours and developing a separation logic to enable VM-modular reasoning about these hypercalls and capture the isolation property.

Relaxed hardware architecture Much of the work on concurrency reasoning in separation logic assumes sequential consistency. However, real-world hardware architectures with relaxed memory models reorder instructions to optimise performance. Consequently, instruction execution may be *out-of-order*, posing a challenge for separation logic, whose consistency largely relies on in-order execution.

This dissertation explores how separation logic techniques can be adapted to reason about realistic relaxed hardware concurrency models while maintaining thread-modular verification. In particular, the focus is on the Arm-A architecture, which features a *very relaxed* memory model, permitting more aggressive instruction reordering. A more detailed introduction to relaxed memory concurrency is provided in Section 2.3.

Thesis The results of this dissertation support the following thesis:

Modern separation logic is an effective tool that achieves modular verification of low-level programs that feature dynamic memory access control or highly relaxed hardware concurrency.

2

Background

In this chapter, we provide a review of separation logic and relaxed memory concurrency, two key topics that form a significant foundation of this dissertation. This serves as a preliminary introduction to the topics covered in the papers of this dissertation. We begin with a brief historical overview of separation logic, highlighting its contributions to modular reasoning for concurrent programs. We then introduce Iris, a state-of-the-art concurrent separation logic framework that unifies many separation logic advancements within a compact yet expressive foundation. Next, we proceed with an introduction to relaxed memory concurrency. Finally, we conclude the chapter with related work.

The chapter is structured as follows:

Section 2.1 provides an overview of key advancements in separation logic. We start by revisiting Hoare logic and discussing its limitations in reasoning about pointers, which were later addressed tractably by separation logic. We then introduce concurrent separation logic (CSL), an extension of separation logic designed for modular verification of concurrent programs. We conclude the section with notable improvements to CSL aimed at achieving a *fiction of separation* to enable even more modular reasoning.

Section 2.2 presents an introduction to Iris, with a particular focus on its higher-order and extensible features. We begin with the Iris base logic, a minimal foundation incorporating step-indexing and higher-order ghost resources. Next, we discuss Iris' language-parametric program logic, which is built upon its base logic.

Section 2.3 introduces the fundamentals of relaxed memory concurrency, which encompasses more realistic concurrency models compared to the idealised sequential consistency upon which most CSL work is based. We begin with a conceptual introduction to the topic, followed by an explanation of different approaches to formulating relaxed memory semantics. We further elaborate by presenting the different styles of the x86-TSO memory model, and touch

upon the memory model of C/C++11 and Arm-A. We conclude the section with a discussion on verification efforts for relaxed concurrency models.

Section 2.4 examines related work on verifying low-level and/or realistic programs, particularly broadly related ones that are not covered in the publications included in Part II.

2.1 Separation Logic

Since its introduction [O'Hearn et al., 2001; Reynolds, 2002], separation logic has become a significant milestone in the field of program verification. It was developed as an enhancement of Hoare logic [Floyd, 1967; Hoare, 1969], with the goal of alleviating the challenges associated with reasoning about programs that manipulate pointers.

Over the years, separation logic has been extensively extended to study various programming language features and type systems. In this section, we focus on aspects of separation logic that are particularly relevant to reasoning about concurrent programs. We refer readers to Charguéraud's habilitation thesis [Charguéraud, 2023] for a comprehensive review on separation logic for sequential programs.

2.1.1 Hoare Logic

Hoare logic is a formal deductive system for reasoning about program correctness. It introduces Hoare triples, which is a logical assertion that specifies the expected behaviour of a program:

$$\{P\}\ e\ \{Q\}$$

A Hoare triple consists of a precondition P, a program e, and a postcondition Q. It asserts that if P holds before executing e, then Q holds upon execution completion. To establish a Hoare triple, one applies axiomatic reasoning rules for the primitive constructs of the programming language, their composition, and the inference of assertions.

To illustrate Hoare logic in practice, we present a selection of its reasoning rules for a simple sequential imperative language, similar to the one presented in Hoare's original work:

$$\text{HL-HT-ASSIGN}\quad \{P[e/x]\}\ x{:=}e\ \{P\} \qquad \text{HL-HT-COMPOSITION}\quad \frac{\{P\}\ e_1\ \{R\} \qquad \{R\}\ e_2\ \{Q\}}{\{P\}\ e_1;e_2\ \{Q\}}$$

$$\text{HL-HT-CONSEQUENCE}\quad \frac{P \to P' \qquad \{P'\}\ e\ \{Q'\} \qquad Q' \to Q}{\{P\}\ e\ \{Q\}}$$

HL-HT-ASSIGN This is the reasoning axiom for assignments. It requires $P[e/x]$ to hold to establish P after the assignment, where $P[e/x]$ is the notation for substituting all free occurrences of x in P with e.

HL-HT-CONSEQUENCE This is the rule of consequence. It allows altering the assertions of a triple via logical deduction. One can update precondition P' to P when P implies P'; and Q' to Q when Q' implies Q.

HL-HT-COMPOSITION This is the rule of sequential composition. It allows combining the triples for programs e_1 and e_2 if the postcondition of the first program R is identical to the precondition of the second triple, when composing the two programs sequentially. This rule is often used alongside the rule of consequence.

Hoare logic has been shown to be effective for reasoning about programs with simple features but encounters difficulties when applied to programs that manipulate pointers. This challenge arises because sound reasoning rules for pointers must account for pointer aliasing, ensuring that updates to the value referenced by a pointer do not affect distinct locations. Alias reasoning also undermines modularity: to soundly compose Hoare triples for two programs that manipulate pointers, one must ensure that the pointers used by the two programs are distinct.

2.1.2 Sequential Separation logic

Separation logic provides a foundational and elegant solution for reasoning about heap-manipulating programs with pointers. It extends Hoare logic with two key operators: *separating conjunction* $*$ (also known as the "star") and *separating implication* $-\!\!*$ (also known as the "magic wand"), which embeds a notion of separation into their semantics comparing to their vanilla counterparts.

Points-to assertions We use the points-to assertion $x \hookrightarrow 1$ to denote that pointer x refers to the integer 1 in the heap. It enables reasoning about pointer updates, for instance, using the SL-HT-PTR-UPD rule:

$$\text{SL-HT-PTR-UPD}\quad \{x \hookrightarrow v\}\ *x := v'\ \{x \hookrightarrow v'\}$$

The assertion $x \hookrightarrow 1 * y \hookrightarrow 2$ implies that the two pointers are distinct; in other words, the heap fragments they represent are *disjoint*. Formally, we have

$$x \hookrightarrow \_ * y \hookrightarrow \_ \vdash x \neq y$$

Framing With this operator, one can formally capture the notion of local reasoning [O'Hearn et al., 2001] through the well-known frame rule:

$$\text{SL-HT-FRAME}\quad \frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}}$$

This rule captures an intuitive but important idea: only assertions representing heap fragments that are touched by the program e (also known as the footprint of e) matter to its proof — one can simply frame out the untouched part, R, from the pre- and postcondition to ease the reasoning. This rule has a profound impact on all subsequent developments in separation logic and is one of the key factors contributing to its success.

Language of resources Separation logic assertions can be viewed as a language of resources, as they revolve around resource reasoning. For instance, for a logic targeting a heap manipulating language, the resources are loosely fragments of a heap h. The heap is represented a finite map from pointers to their values, which forms the basis for defining the semantics of logical assertions as follows:

$$\begin{aligned}
[\![x \hookrightarrow v]\!](h) &\triangleq x \in \mathsf{dom}(h) \wedge h(x) = v \\
[\![P * Q]\!](h) &\triangleq \exists h_1 \cup h_2 = h.\, [\![P]\!](h1) \wedge [\![Q]\!](h2) \wedge h_1 \# h_2 \\
[\![P \mathrel{-\!\!*} Q]\!](h) &\triangleq \forall h' \# h.\, [\![P]\!](h) \wedge [\![Q]\!](h \cup h')
\end{aligned}$$

The points-to $x \hookrightarrow v$ concerns the heap cell x of h. The semantics of separating conjunction requires that the heap fragment satisfying $P * Q$ can be split into two smaller and disjoint fragments, h_1 and h_2, such that h_1 satisfies P and h_2 satisfies Q. The separating implication intuitively represents the resources required to obtain the resources of Q, given the resources of P.

Linear and affine logics Generally, separation logics are categorised into two kinds, depending on if they admit the following weakening rule:

$$\frac{}{P * Q \vdash P}\ \textsc{sl-weaken}$$

This rules allows freely "throwing away" resources. Logics admitting this rule are affine, in which the semantics of assertions are monotone. It means that if some resource satisfies an assertion, then resources larger than it also satisfy the assertion. Logics not admitting this rule are linear, in which assertions tracks resources precisely. Linear logics are suitable for reasoning about memory leaks. For instance, to show an empty postcondition, one has to show that the program indeed frees all the allocated memory.

2.1.3 Concurrent Separation Logic

Concurrent Separation Logic (CSL) [Brookes, 2007; O'Hearn, 2007] is a significant extension of separation logic that fully realises its potential for reasoning about concurrent programs. CSL represents a major breakthrough in addressing the long-standing challenge of compositional reasoning for shared-memory concurrency. One

key contribution is its parallel decomposition rule:

$$\text{CSL-DISJ-PAR}\quad \frac{\{P_1\}\ e_1\ \{Q_1\} \qquad \{P_2\}\ e_2\ \{Q_2\}}{\{P_1 * Q_1\}\ e_1 \mid\mid e_2\ \{Q_1 * Q_2\}}$$

CSL-DISJ-PAR allows reasoning about two parallel threads e_1 and e_2 independently, provided that the heap fragments accessed by each thread are *disjoint*. This form of concurrency, where threads operate without interfering others, is referred to as *disjoint concurrency*.

Resource sharing

To reason about shared-state concurrency in general, a more flexible rule is required: one that allows multiple threads to access shared heap regions for communication.

Achieving this while remaining thread-modular is challenging, as it requires the logic to account for *global interference*. That is, when reasoning locally about a shared heap cell in one thread, the logic must also consider how other threads may modify the same cell concurrently.

The initial work on CSL [Brookes, 2007; O'Hearn, 2007] addresses this problem in coarse-grained concurrency (i.e., well-locked programs) using *resource invariants*. Resource invariants ensure that shared resources become available locally to a thread once it enters a critical section, where interference from other threads is prevented. To make the use of invariants sound, the parallel decomposition rule has additional constrains to handle interference, which makes it rather intractable to use.

Parkinson et al. [2007] demonstrates how to reason about *fine-grained* concurrent programs using invariants. Rather than relying on critical sections for sound resource sharing, their logic identifies *atomic* operations, such as compare-and-swap (CAS), as fine-grained alternatives to critical sections.

$$\text{CSL-INV-ATOMIC}\quad \frac{\Gamma \vdash \{P * I\}\ e\ \{Q * I\} \qquad \text{atomic}(e)}{\Gamma, I \vdash \{P\}\ e\ \{Q\}}$$

The rule CSL-INV-ATOMIC enables the use of an invariant resource I within the execution of an atomic operation e.

An ownership perspective O'Hearn [2007] conceptualises resource sharing between threads as *ownership transfer*. The assertion $x \hookrightarrow v$ is read as the ownership of the heap cell storing value v; which allows the owning thread to modify the cell. Resource invariants effectively facilitate the transfer of ownership between threads. Bornat et al. [2005] extend this interpretation by introducing a notion of *permissions*. The assertion $x \hookrightarrow_q v$, where the fraction $0 < q \leq 1$, denotes q shares of the ownership. When $q = 1$, the thread has full ownership of the cell therefore the permission to read, write, and deallocate it. Otherwise, the thread has partial

ownership, thus read-only permission. Allocations always give full permission. Permissions can be split and combined as follows:

CSL-PT-PERM
$$x \hookrightarrow_{q_1} v * x \hookrightarrow_{q_2} v \dashv\vdash x \hookrightarrow_{q_1+q_2} v * (q_1 + q_2 \leq 1)$$

Modular Reasoning for Fine-Grained Concurrency

Extensive follow-up work [Dinsdale-Young et al., 2013, 2010; Dodds et al., 2009; Feng, 2009; Nanevski et al., 2014; Svendsen and Birkedal, 2014; Vafeiadis and Parkinson, 2007] introduces various abstraction techniques that enhance the modularity and flexibility of CSL, making it more effective for reasoning about fine-grained concurrency.

Concurrent abstract predicates Dinsdale-Young et al. [2010] introduce concurrent abstract predicates (CAP) to facilitate reasoning about fine-grained concurrent modules (e.g., a CAS-based lock). The abstract predicates provide a *fiction of separation*. By abstracting over the internal interference of the module, the clients can use the module's predicate as if they owned the module and apply its CAP specification in a thread-local manner. This is achieved by requiring the abstract predicate of a module to remain preserved under a predefined collection of *abstract actions* of the module that may affect its heap fragments. These heap fragments are shared among the threads (the module's clients) via invariants, with *protocols* specifying how those abstract actions mutate the shared heap regions. To prove a CAP specification for a module, one must account for all actions that other threads may perform concurrently on the shared regions, ensuring correctness under all possible interference scenarios.

For instance, a CAP specification that captures the behaviour of a classic CAS-based lock implementation is as follows:

CAP-LOCK
$$\{\mathsf{isLock}(x)\}\ \texttt{lock(x)}\ \{\mathsf{isLock}(x) * \mathsf{Locked}(x)\}$$

CAP-UNLOCK
$$\{\mathsf{isLock}(x) * \mathsf{Locked}(x)\}\ \texttt{unlock(x)}\ \{\mathsf{isLock}(x)\}$$

The protocol for the lock permits two abstract actions corresponding to the two operations: `lock` and `unlock`, which are tracked using permission assertions: the assertion $\mathsf{Locked}(x)$ carries the exclusive unlock permission, which a thread obtains after performing `lock` as specified in CAP-LOCK, and is consumed by `unlock` in CAP-UNLOCK. The abstract predicate $\mathsf{isLock}(x)$ carries the lock permission, which grants the permission to update the value of the shared heap cell referenced by pointer x during the verification of the `lock` procedure. The `lock` procedure employs a CAS operation to read and update the value of x atomically, performing the `lock` action following the protocol. During this CAS update, the ownership of the shared heap cell is available temporarily from the invariant via a CAP variation of CSL-INV-ATOMIC.

The higher-order extensions of CAP, HOCAP [Svendsen et al., 2013] and iCAP [Svendsen and Birkedal, 2014], generalise the CAP idea to higher-order separation logic. They provide much more power of abstraction, which enables even more general and modular specifications for fine-grained concurrent modules. For instance, the `lock` procedure of the same CAS-based lock can now be specified more generically in HOCAP-style as follows:

HOCAP-LOCK
$\{\mathsf{isLock}(x, R)\}\ \mathtt{lock(x)}\ \{\mathsf{isLock}(x, R) * \mathsf{Locked}(x) * R\}$

The new higher-order predicate isLock is additionally parameterised by a proposition R, which remains unknown when verifying the lock implementation. This predicate can be interpreted as that the lock implemented at location x protects the resource R. The rule HOCAP-LOCK allows obtaining the ownership of R upon acquiring the lock. The resource R can represent any assertion, which enables the construction of layered abstractions and recursive predicates. These are essential for modular verification of more advanced concurrent modules beyond CAS-based locks.

User-specified resources The Views framework [Dinsdale-Young et al., 2013] provides a simple yet powerful meta theory that unifies the core ideas of many results in CSL. It enhances the flexibility of CSL by introducing a notion of *views*. Views are semigroups equipped with a composition operator that provides a generalised notion of separation. Intuitively, views capture a thread's knowledge of the physical state and its permissions to update the state. The composition of views corresponds to the aggregation of knowledge and permissions. To ensure soundness, any update to a thread's view must preserve the views held by other threads. The Views framework can be instantiated with user-defined views that introduce appropriate abstractions over the physical state, effectively enabling *user-specified resources*. The standard points-to assertions used to represent heap cells are just a simple instance of views with minimal abstraction over the physical heap.

Linearisability and atomic triples TaDA [da Rocha Pinto et al., 2014] combines the ideas of linearisability [Herlihy and Wing, 1990] and CAP to provide a *fiction of atomicity* for non-atomic concurrent operations. Linearisability is a strong correctness criterion for concurrent modules. It requires that all concurrent calls to a module's operations must have a *linearisation point*, and the calls appear as if they occur *atomically* at their linearisation points.

TaDA captures this notion of *logical atomicity* through *atomic triples*. The atomic triple $\langle P\rangle\ e\ \langle Q\rangle$ allows the clients of e to update P to Q with invariant resource I, as if e were an atomic operation:

TADA-INV-ATOMIC
$$\frac{\Gamma \vdash \langle P * I\rangle\ e\ \langle Q * I\rangle}{\Gamma, I \vdash \langle P\rangle\ e\ \langle Q\rangle}$$

To establish such a triple, it must be shown that although operation e may consist of multiple execution steps, P transitions to Q atomically at its linearisation point. At all other points during the execution of e, P and Q must remain unchanged. For instance, below is a possible specification of the `lock` operation of CAS lock formulated using an atomic triple:

$$\langle b.\, \mathsf{LockState}(x, b)\rangle\ \texttt{lock(x)}\ \langle \mathsf{LockState}(x, \texttt{true})\rangle$$

The predicate $\mathsf{LockState}(x, b)$ means that the lock x is at an abstract boolean state b representing if the lock is taken. The state b is bounded in the precondition, instead of being quantified outside of the triple, to denote that b is the state of the lock that one can only know at the linearisation point of `lock(x)`. This is because other calls to the lock module may change the state concurrently during `lock(x)` before the linearisation point. The `lock` operation is expected to change the state to `true`, as reflected in the postcondition, indicating the lock has been successfully taken.

TaDA's atomic triple is essentially equivalent in expressive power to HOCAP-style specifications. Both approaches leverage user-defined ghost resources and have become the de facto standards on strong specifications for fine-grained concurrent modules.

An alternative approach to formulate linearisability is trace-based. The FCSL logic [Nanevski et al., 2014] supports using auxiliary time-stamped histories to specify concurrent modules with regular Hoare triples [Sergey et al., 2015b]. Histories are tracked with resources, such that the fragmental and subjective views of histories can be separated and used by clients. This idea is then generalised to specify non-linearisable concurrent modules [Sergey et al., 2016].

2.2 Iris: A Higher-Order Concurrent Separation Logic Framework

Iris [Jung et al., 2016, 2018b, 2015; Krebbers et al., 2017a] is a state-of-the-art higher-order concurrent separation logic framework that integrates and refines many of the advances in separation logic discussed in Section 2.1. It unifies key features of prior CSLs for modular concurrent reasoning through two fundamental yet powerful primitives: *monoids* and *invariants*. Since its introduction, Iris has been successfully applied to the studies of various advanced programming language features and type systems, as well as the verification of sophisticated sequential and concurrent modules, with over a hundred publications.

Iris supports advanced separation logic features and foundational proofs, which is the key reason the results of this dissertation are based on it. In this section, we introduce the core features of the framework. This section is structured as follows:

Section 2.2.1 introduces the base logic of Iris, which serves as the core foundation upon which the rest of its features are implemented.

Section 2.2.2 presents the key features of Iris, including impredicative invariants for resource sharing and the language-agnostic program logic for verification.

2.2.1 Base Logic

Iris is structured as a two-layer hierarchy: the *base logic*, which supports higher-order ghost resources and guarded recursion, and the *language-agnostic program logic* extension, built upon the base logic. The logical constructs of the program logic, including invariants and Hoare triples, are derived from the base logic. The assertion language of Iris' base logic is depicted in Figure 2.1. All Iris propositions are of type *iProp*.

$$P, Q \in \mathit{iProp} \triangleq \ulcorner \varphi \urcorner \mid \mathsf{True} \mid \mathsf{False} \mid P \Rightarrow Q \mid P \wedge Q \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\!*} Q \\ \mid \exists x.\, P \mid \forall x.\, Q \mid \triangleright P \mid \Box P \mid \mathsf{Own}\,(a) \mid \pmb{\mathrel{\dot{\Rightarrow}}} P \mid \cdots$$

Figure 2.1: The grammar of Iris base logic

Iris base logic is a:

separation logic It includes the standard separation logic connectives: separating conjunction $*$ and separating implication $\mathrel{-\!\!*}$.

resource-aware logic It introduces $\mathsf{Own}\,(a)$ as an assertion for ownership of ghost resource a; the basic update modality $\pmb{\mathrel{\dot{\Rightarrow}}}$ ensures sound resource updates; the persistently modality $\Box$ asserts the duplicable portion of resources.

higher-order and step-indexing logic It supports higher-order quantification ($\forall$ and $\exists$) over Iris propositions. It also allows recursive predicates and higher-order ghost states via the later modality $\triangleright$.

affine logic It admits the weakening rule SL-WEAKEN.

Soundness

The soundness (or consistency) of Iris base logic is established through its semantic model that interprets all Iris base logic assertions. The logic in which this semantic model is formalised is referred to as the *meta-logic*. Iris' meta-logic is the logic of Coq/Rocq, as it is mechanised and proven sound within the Coq/Rocq proof assistant. Iris allows embedding meta-level propositions φ using the notation $\ulcorner \varphi \urcorner$, though this is often omitted when it is clear that φ belongs to the meta-logic. Such propositions are classified as *pure propositions*, meaning they do not involve any Iris-specific features such as resources or modalities. The soundness of the Iris base logic can be formulated in terms of a pure proposition φ, as stated in Theorem 2.2.1.

Theorem 2.2.1 (Soundness of Iris base logic)**.** *Given the following Iris entailment for meta level proposition φ behind Iris' modalities dealing with resource update ($\pmb{\mathrel{\dot{\Rightarrow}}}$) and higher-orderness ($\triangleright$) iterated n times*

$$\mathsf{True} \vdash (\pmb{\mathrel{\dot{\Rightarrow}}} \triangleright)^n \ulcorner \varphi \urcorner$$

then φ holds in the meta-logic.

Step-Indexing and Later Modality

Iris is a step-indexing logic that supports higher-order ghost resources and guarded recursion — two key features to make it expressive and extensible. The step-indexing of Iris is modelled using Ordered Families of Equivalences (**OFE**). As the name suggests, an **OFE** consists of a set of elements T equipped with a family of equivalence relations $\{\overset{n}{=} \subseteq (T \times T) \mid n \in \mathbb{N}\}$ on T. Intuitively, the natural number index n measures the strength of the equivalence: the larger the index, the stronger the equivalence. This can also be interpreted as a measure of how indistinguishable two elements related by $\overset{n}{=}$ are, where n represents the number of computational steps required to distinguish them.

Very loosely speaking, Iris propositions are defined as step-indexed predicates over its model of resources. Step-indexed propositions (*SProp*) form a **OFE**: they are downward-closed sets of step indices (natural numbers), meaning that if a proposition holds at some step index, it must also hold at all smaller step indices.

$$\mathit{SProp} \triangleq \{s \in 2^{\mathbb{N}} \mid \forall m \leq n.\, n \in s \Rightarrow m \in s\}$$

The Iris base logic employs a *later modality* $\triangleright$ to abstract over tedious explicit step-indexing reasoning. The assertion $\triangleright P$ means that P holds one computation step later. Semantically, if P holds for step indices 0 through n, then $\triangleright P$ holds for step indices 0 through $n+1$. The key rules for the later modality are as follows:

Iris-Later-intro
$$P \vdash \triangleright P$$

Iris-Later-löb
$$(\triangleright P \Rightarrow P) \vdash P$$

Iris-Later-mono
$$\frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

Iris-Later-intro As per the semantics of $\triangleright$, this rule allows concluding that P holds one step later, provided that P holds at the current step.

Iris-Later-löb The later modality enables *guarded recursion*, allowing the guarded fixpoint of recursive predicate definitions, where recursive occurrences are placed behind a later modality. This rule states that a recursive predicate P can be established if one can show P, assuming that its recursive occurrence $\triangleright P$ holds. This is useful when reasoning about guarded recursive predicates.

Iris-Later-mono This rule asserts that the later modality preserves the logical entailment. It is often used to eliminate a later modality from assumptions.

Ghost resources

Iris unifies resources representing physical states and ghost states. It uses a notion of resource algebra (RA) as the model of (first-order) resources.

Resource algebra A resource algebra is a collection of elements (M) with three operations:

Composition $\cdot : M \to M \to M$ is a binary operator for composition of elements. The composition operation is associative and commutative.

Core $|-| : M \to M^?$ is a partial function that returns a core of an element, where $M^? \triangleq M + \{\bot\}$. A core of element a is a fragment of a that is duplicable: $|a| \cdot a = a$. If a does not have a duplicable core, then $|a| = \bot$. The core operation is idempotent: $||a|| = |a|$.

Validity $\mathcal{V} : M \to \mathit{Prop}$ is a validity predicate identifying the valid elements of M. Composition is only meaningful on valid elements. If an element is valid, then all of its components are also valid.

There are two important derived notations:

Extension order $a \preccurlyeq b$ means there is a $c \in M$ such that $a \cdot c = b$. The core operation preserves extension: $|a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b|$.

Frame-preserving update $a \rightsquigarrow B$ means that element a can be updated to $B \subseteq M$ if, for every frame of a — elements whose composition with a is valid — there exists a $b \in B$ whose composition with the frame is also valid. Formally:

$$a \rightsquigarrow B \triangleq \forall c \in M^?.\, \mathcal{V}(a \cdot c) \Rightarrow \exists b \in B.\, \mathcal{V}(b \cdot c)$$
$$a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$$

The model of higher-order resources generalises RA using step-indexing. Instead of providing its definition, we present examples of higher-order RA later in this section.

Resource Ownership and Update

Iris' primitive assertion $\mathsf{Own}\,(a : M)$ expresses ownership of a valid element a within some RA M. Iris provides several connectives, elaborated below, that interact with this ownership assertion and are governed by the following laws:

IRIS-OWNM-OP
$$\mathsf{Own}\,(a) * \mathsf{Own}\,(b) \dashv\vdash \mathsf{Own}\,(a \cdot b)$$

IRIS-OWNM-PERS
$$\mathsf{Own}\,(a) \vdash \Box\, \mathsf{Own}\,(|a|)$$

IRIS-OWNM-BUPD
$$\frac{a \rightsquigarrow B}{\mathsf{Own}\,(a) \vdash \dot{\Rightarrow} \exists b \in B.\, \mathsf{Own}\,(b)}$$

IRIS-OWNM-OP This rule means owning resources a and b separately, as per the separating conjunction, is equivalent to owning their composition.

IRIS-OWNM-PERS This rule means owning resource a implies owning its core $|a|$ which is persistent knowledge, as per the persistently modality $\Box$.

IRIS-OWNM-BUPD This rule says one may update resource a to any $b \in B$ with the basic update modality $\dot{\Rightarrow}$ if the update is frame-preserving. The modality only indicates that the update is *allowed*, but not completed. To complete the update and use the updated resources freely, one has to remove the modality.

Persistently modality and persistent propositions The persistently modality is a modality that is easy to eliminate but hard to introduce.

$$\textsc{Iris-pers-idemp}\quad \Box\Box P \dashv\vdash \Box P \qquad \textsc{Iris-pers-elim}\quad \Box P \vdash P \qquad \textsc{Iris-pers-intro}\quad \frac{\Box P \vdash Q}{\Box P \vdash \Box P}$$

Iris-pers-idemp The persistently modality is idempotent.

Iris-pers-elim This rules allows removing the persistently modality from propositions at any time. Combined with Iris-pers-idemp, one can remove arbitrary number of it.

Iris-pers-intro This rules allows introducing the persistently modality to Q if one can show Q with only assumptions guarded by the modality. It is usually applied to remove the modality from the goal.

The persistently modality is used to define a kind of propositions called *persistent proposition*. P is persistent if $P \vdash \Box P$, which means the proposition does not own any non-duplicable portion of resources. Such propositions represent knowledge that does not change once established, and are freely duplicable. For instance, pure propositions and the assertion Own ($|a|$) are persistent.

Basic update modality and resource update The basic update modality is straightforward to introduce but difficult to eliminate, in contrast to the persistently modality. It is called the *basic* update modality because Iris also provides a more powerful *fancy* update modality which is built upon the basic one and elaborated in Section 2.2.2.

$$\textsc{Iris-bupd-idemp}\quad \pmb{\mathrel{\dot{\Rightarrow}}} \pmb{\mathrel{\dot{\Rightarrow}}} P \dashv\vdash \pmb{\mathrel{\dot{\Rightarrow}}} P \qquad \textsc{Iris-bupd-intro}\quad P \vdash \pmb{\mathrel{\dot{\Rightarrow}}} P \qquad \textsc{Iris-bupd-elim}\quad \frac{P \vdash \pmb{\mathrel{\dot{\Rightarrow}}} Q}{\pmb{\mathrel{\dot{\Rightarrow}}} P \vdash \pmb{\mathrel{\dot{\Rightarrow}}} P}$$

Iris-bupd-idemp The basic update modality is idempotent.

Iris-bupd-intro This rules allows introducing the basic update modality at any time.

Iris-bupd-elim This rules allows eliminating a basic update modality from the assumption P if the goal has a basic update modality. In a sense, the modality on the goal denotes a permission to perform the resource updates in P. Since updating resources introduces an update modality (Iris-ownM-bupd), this rules is used to remove the introduced modality to use the updated resources.

Basic view shift The basic view shift $P \mathrel{\dot{\Rightarrow}} Q$ is the persistent fact that P can be updated to Q.

$$P \mathrel{\dot{\Rightarrow}} Q \triangleq \Box(P \mathrel{-\!\!*} \pmb{\mathrel{\dot{\Rightarrow}}} Q)$$

Multiple RAs Iris supports the parallel use of multiple resource algebras (RAs) by constructing a partial function RA that indexes each used RA M with a unique identifier i. Additionally, Iris allows multiple instances of the same RA, which is achieved by constructing another partial function RA for each M_i, where every instance of M_i is indexed by a ghost name $\gamma \in \mathsf{GName}$.

The notation $\boxed{a : M_i}^{\gamma}$ asserts the ownership of the resource a from RA M_i under the ghost name γ, defined using the ownership assertion. The RA M_i is often omitted when it is unambiguous from the context.

The following essential rules can be derived for this assertion from the rules of the basic update modality:

IRIS-OWNG-OP
$$\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$$

IRIS-OWNG-ALLOC
$$\frac{\mathcal{V}_{M_i}(a)}{\mathsf{True} \Rrightarrow \exists \gamma : \mathsf{GName}. \boxed{a : M_i}^{\gamma}}$$

IRIS-OWNG-UPDATE
$$\frac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \Rrightarrow \boxed{b}^{\gamma}}$$

IRIS-OWNG-OP This rule says the resources governed by the same ghost name γ (indicating they belong to the same RA instance) can be combined or split following the composition of the RA.

IRIS-OWNG-ALLOC This rule allows allocating the resource a of RA M_i with a fresh ghost name γ if a is valid.

IRIS-OWNG-UPDATE This rules is derived from the rule of basic update and the definition of view shift. It allows resource a to update to b if the update is frame-preserving, which is crucial for the soundness of resource update.

Higher-order ghost resources Iris supports *higher-order* ghost resources by extending the resource model from RAs to CMRAs. CMRAs integrate step-indexing into RAs by requiring the element collection M to form an **OFE**. Consequently, the assertion $\boxed{a}^{\gamma}$ now takes elements from CMRAs. This extension allows Iris propositions to be referenced within ghost resources, effectively making ghost resources *higher-order*.

For instance, Iris includes a savedProp CMRA, implemented using the *agreement* constructor (elaborated below) which maps Iris propositions to a CMRA. The assertion $\boxed{\mathsf{savedProp}(q, P)}^{\gamma}$ stores a fractional share $q \in (0, 1]$ of the Iris proposition P as a ghost resource, and follows two key rules:

IRIS-SAVEDPROP-AG
$$\boxed{\mathsf{savedProp}(q_1, P)}^{\gamma} * \boxed{\mathsf{savedProp}(q_2, Q)}^{\gamma} \vdash (q_1 + q_2 \leq 1) * \triangleright (P = Q)$$

IRIS-SAVEDPROP-UPD
$$\boxed{\mathsf{savedProp}(1, P)}^{\gamma} \Rrightarrow \boxed{\mathsf{savedProp}(1, Q)}^{\gamma}$$

IRIS-SAVEDPROP-AG This rule allows concluding the equality of two stored Iris propositions. The equality is behind a later modality for the soundness of higher-order ghost resources.

IRIS-SAVEDPROP-UPD This rule allows updating the full share of the saved proposition P to an arbitrary Q.

Common (CM)RA constructors

Iris provides some basic (CM)RA constructors. These constructors act as building blocks for more sophisticated ghost resources, supporting the diverse needs of program verification. We present some representative ones below.

Agreement CMRA For any **OFE** T, $Ag(T)$ is a CMRA with a injection function $\mathsf{ag}(x) \triangleq \{x\}$ that maps $x \in T$ to $Ag(T)$. Intuitively, this function maps x of T to a CMRA which contains only replicas of x. Indeed, we have $\mathsf{ag}(x) \cdot \mathsf{ag}(x) = \mathsf{ag}(x)$. A frame-preserving update to $\mathsf{ag}(x)$ is not possible, as the number of replicas (the frames) is unknown. This CMRA allows embedding Iris propositions, and savedProp is implemented using it.

Product CMRA Given two CMRAs M_1 and M_2, $M_1 \times M_2$ is a CMRA whose elements are pairs: $\{(x, y) \mid x \in M_1 \land y \in M_2\}$. The CMRA operations are pointwise.

Fraction RA Fraction RA is widely used to pair with some CMRA to track its shares. For instance, it is employed to implement fractional permissions [Bornat et al., 2005].

$$\begin{aligned} M &\triangleq \mathsf{frac}((0,1]) \mid \lightning \\ \mathcal{V}(a) &\triangleq a \neq \lightning \\ \mathsf{frac}(q_1) \cdot \mathsf{frac}(q_2) &\triangleq \mathsf{frac}(q_1 + q_2) \text{ when } (q_1 + q_2) \leq 1 \end{aligned}$$

The elements of this RA are fractions and a bottom element. Fractions compose through addition.

Authoritative CMRA Given a CMRA M, the CMRA $Auth(M)$ provides two views (the Iris adaptation of the views in the Views framework) for some $x \in M$: the *authoritative view* $\bullet x$ and the *fragmental view* $\circ x$. These two views follow the following key rules:

IRIS-AUTH-AUTH-OP
$\mathcal{V}(\bullet x \cdot \bullet y) \Rightarrow \mathsf{False}$

IRIS-AUTH-FRAG-OP
$(\circ x \cdot \circ y) = \circ(x \cdot y)$

IRIS-AUTH-BOTH-OP
$\mathcal{V}(\bullet x \cdot \circ y) \Leftrightarrow y \preccurlyeq x \land \mathcal{V}(x)$

IRIS-AUTH-BOTH-UPD
$$\frac{(x, y) \rightsquigarrow_l (x', y')}{\bullet x \cdot \circ y \rightsquigarrow \bullet x' \cdot \circ y'}$$

IRIS-AUTH-AUTH-OP This rule states that the authoritative view is *exclusive*, meaning it represents the unique and complete knowledge of the resource $x \in M$.

IRIS-AUTH-FRAG-OP This rule ensures that fragmental views can be composed by combining the elements they carry. An authoritative view may have multiple distinct fragmental views, and this rule ensures that their composition remains a valid fragmental view.

IRIS-AUTH-BOTH-OP This rule establishes that fragmental views provide *partial knowledge* of x: all valid fragmental views must be contained within the corresponding authoritative view.

IRIS-AUTH-BOTH-UPD This rule allows simultaneous updates to both views when the frame-preserving update is a *local update*, denoted by $\rightsquigarrow_l$. Intuitively, the local update ensures that any fragmental frame whose composition with y results in x is preserved: its composition with y' must result in x'. This is trivially satisfied when y has no frames or when the update from x to x' affects only the portion of the view disjoint from y.

The Authoritative CMRA captures the *fiction of separation* and is widely used for modular reasoning in Iris. An example of this CMRA applied to a finite heap is presented later in this section.

2.2.2 Program Logic

Iris provides a generic program logic framework built on top of its base logic. This framework is *language-parametric*.

Invariants and fancy update modality

Iris supports *impredicative invariants*, denoted as $\boxed{P}^{\iota}$, for resource sharing in the program logic. Here, P is an arbitrary Iris proposition (which can itself be another invariant), and $\iota \in \mathsf{InvName}$ represents the invariant's unique name. Invariants are persistent, meaning they can be freely shared among threads. Threads can open invariants (and violate them) for an atomic step, as specified in IRIS-HT-INV.

$$\frac{\vdash \{\triangleright I * P\}\ e\ \{\triangleright I * Q\}_{\mathcal{E}\setminus\{\iota\}} \qquad \mathsf{atomic}(e) \qquad \iota \in \mathcal{E}}{\boxed{I}^{\iota} \vdash \{P\}\ e\ \{Q\}_{\mathcal{E}}}\ \text{IRIS-HT-INV}$$

To ensure that each invariant is opened at most once at any given time, Iris employs *masks* (collections of invariant names) to track which invariants are permitted to be opened. The Hoare triple in Iris is parameterised by a mask $\mathcal{E}$. When an invariant is opened, its name is removed from $\mathcal{E}$ to prevent unsound repeated openings. For soundness, the associated invariant resource I is guarded by a *later modality*, meaning it becomes accessible only after at least one computation step. However, in practice, this does not usually pose an issue, as the later modality can be eliminated from a class of propositions that do not refer to invariants without requiring a computation step.

Fancy update modality Under the hood, the opening and closing of invariants are managed by the *fancy update modality* ${}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2}$, which extends the basic update modality with two masks. It is implemented using the basic update modality combined with ghost resources. The assertion ${}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2} P$ expresses that, prior to the resource update, all invariants whose names are in the opening mask $\mathcal{E}_1$ remain closed (hence, they may be opened), and after updating to P, the closing mask $\mathcal{E}_2$ defines the set of invariants that remain closed. By specifying the masks before and after an update, one can control which invariants may be opened or which must be closed. $\mathcal{E}_1 = \mathcal{E}_2$ ensures that the set of closed invariants remains unchanged after the update. However, this does not preclude invariants in $\mathcal{E}_1$ from being temporarily opened and then closed. The shorthand notation $\Rightarrow_{\mathcal{E}_1}$ is used for this case.

The non-mask-changing fancy update modality $\Rightarrow_{\mathcal{E}}$ serves as a drop-in replacement for the basic update modality and satisfies all its corresponding rules. The mask-changing one additionally adheres to the following key rules:

Iris-fupd-trans
$${}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2} {}_{\mathcal{E}_2}\Rightarrow_{\mathcal{E}_3} P \vdash {}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_3} P$$

Iris-fupd-intro
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{P \vdash {}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2} {}_{\mathcal{E}_2}\Rightarrow_{\mathcal{E}_1} P}$$

Iris-fupd-frame
$$Q * {}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2} P \vdash {}_{\mathcal{E}_1 \uplus \mathcal{E}_f}\Rightarrow_{\mathcal{E}_2 \uplus \mathcal{E}_f} (Q * P)$$

Iris-fupd-trans This rule allows chaining mask changes, which is useful when the goal's mask do not match those in the assumption.

Iris-fupd-intro This rule allows introducing masks. This is often implicitly use when removing the modality from the goal.

Iris-fupd-frame This rule allows framing away masks that are not useful to show the goal.

Fancy update modality also has notations for view shifts:

$$P \;{}_{\mathcal{E}_1}\Rrightarrow_{\mathcal{E}_2} Q \triangleq P \twoheadrightarrow {}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2} Q$$
$$P \;{}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2} Q \triangleq \Box(P \;{}_{\mathcal{E}_1}\Rrightarrow_{\mathcal{E}_2} Q)$$
$$P \;\Rightarrow_{\mathcal{E}} Q \triangleq P \;{}_{\mathcal{E}}\Rightarrow_{\mathcal{E}} Q$$

The difference between the first two view shifts is that the first can only be applied once, while the second is persistent.

The following rules illustrate how the fancy update modality interacts with invariants:

Iris-inv-alloc
$$\triangleright P \;\Rightarrow_{\emptyset} \boxed{P}^{\mathcal{N}}$$

Iris-inv-open
$$\frac{\mathcal{N} \in \mathcal{E}}{\boxed{P}^{\mathcal{N}} \;{}_{\mathcal{E}}\Rightarrow_{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \;{}_{\mathcal{E}\setminus\mathcal{N}}\Rrightarrow_{\mathcal{E}} \mathsf{True})}$$

Iris-inv-alloc This rule allows the allocation of an invariant under a namespace $\mathcal{N}$, instead of a specific name. In Iris, invariants are identified by namespaces

$\mathcal{N}$, which are sets of invariant names. This approach generalises the concrete invariant name ι used previously, as reasoning typically concerns disjointness and inclusion among sets of invariant names. This generalisation is necessary to allow the flexible allocation of invariants within arbitrary namespaces. The allocation is performed using an empty mask $\emptyset$.

IRIS-INV-OPEN This rule permits opening an invariant with the namespace $\mathcal{N}$, provided that the view shift mask includes it. The associated invariant resource is made available as $\triangleright P$. Upon opening the invariant, the mask is updated to $\mathcal{E} \setminus \mathcal{N}$. To restore the original mask and reestablish the invariant, the invariant resource $\triangleright P$ must be returned, as required by the view shift.

The Iris invariant is formally defined as follows:

$$\boxed{P}^{\mathcal{N}} \triangleq \Box \forall \mathcal{E}.\ {}_{\mathcal{E}}\Rightarrow_{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \Rightarrow_{\mathcal{E}\setminus\mathcal{N}} {}_{\mathcal{E}}\!\!*\ \mathsf{True})$$

Hoare triples and Weakest preconditions

Hoare triples in Iris manage invariants by incorporating masks via the fancy update modality. They are defined using *weakest preconditions* as follows:

$$\{P\}\ e\ \{Q\}_{\mathcal{E}} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\ e\ \{Q\})$$

$\mathsf{wp}_{\mathcal{E}}\ e\ \{Q\}$ represents the *weakest* precondition that ensures that the postcondition Q holds upon termination of e. It is parameterised by the invariant mask $\mathcal{E}$, so that invariants whose namespaces included in the mask can be opened.

Iris provides a default, *language-independent* definition of weakest preconditions for partial correctness. To instantiate it for a specific language, one needs to provide a small-step operational semantics, including a notion of physical shared state and thread-local reductions that mutate the state. To link the semantics with the program logic, one needs to provide a logical predicate called *state interpretation* to interpret the physical state using the ghost resources of a customised *ghost theory*. This ghost theory defines rules governing the interaction between the state interpretation and logical assertions, which are also implemented using the ghost resources.

To elaborate, we present key elements of the weakest precondition definition, specialised for a toy heap-based language. For a comprehensive description of the full definition, we refer readers to the Iris papers [Jung et al., 2016, 2018b, 2015]. The shared physical state of this toy language is a heap σ, represented as a finite map from locations to values. Its operational semantics consists of two levels of reduction relations.

Thread-local reduction This reduction relation form $\langle e, \sigma\rangle \rightarrow \langle e', \sigma'\rangle$ describes how one thread may mutate the heap atomically. The semantics of the three heap

operations of the toy language are specified as follows:

$$\textsc{Opsem-tl-load}\ \frac{\sigma[x] = v}{\langle !x, \sigma\rangle \to \langle v, \sigma\rangle} \qquad \textsc{Opsem-tl-store}\ \langle x := v, \sigma\rangle \to \langle (), \sigma[x \mapsto v]\rangle$$

$$\textsc{Opsem-tl-alloc}\ \frac{x \notin \mathsf{dom}(\sigma)}{\langle \mathsf{ref}(v), \sigma\rangle \to \langle x, \sigma \cup \{x \mapsto v\}\rangle}$$

Opsem-tl-load This rule for load returns the value of location x in the heap.

Opsem-tl-store This rule for store update the value of x in the heap.

Opsem-tl-alloc This rule for allocation returns a fresh location x that stores v.

Thread-pool reduction The toy language has fixed number of threads. Its thread-pool reduction is of the form $\langle \vec{e}, \sigma\rangle \to_{\mathsf{tp}} \langle \vec{e}', \sigma'\rangle$ where $\vec{e}$ and $\vec{e'}$ are the programs of all threads (*i.e.*,the thread pool). Concurrency (under sequential consistency) is modelled by non-deterministically selecting a thread to perform a thread-local reduction step with the rule below:

$$\textsc{Opsem-tp}\ \frac{\langle e, \sigma\rangle \to \langle e', \sigma'\rangle}{\langle \vec{e}; e; \vec{e}', \sigma\rangle \to_{\mathsf{tp}} \langle \vec{e}; e'; \vec{e}', \sigma'\rangle}$$

The transitive closure of thread-pool reduction represents the interleaving of thread-local reductions.

Definition The overall structure of the weakest precondition $\mathsf{wp}_{\mathcal{E}}\, e\, \{Q\}$ for the language consists of two cases corresponding to whether the execution of the program e has terminated or not.

In the terminating case, we require the postcondition Q holds on e after a resource update:

$$\mathsf{IsVal}(e) \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} Q(e)$$

The termination of a program is determined by the predicate IsVal over program e: it holds when e is a value thus cannot be further reduced.

If e remains a reducible expression, a universally quantified physical shared heap σ is assumed, along with its *state interpretation* $\mathsf{SI}(\sigma)$ that provides an abstract logical view of σ using the ghost theory. The weakest precondition requires satisfying two obligations while opening invariants permitted by $\mathcal{E}$:

$$\neg\mathsf{IsVal}(e) \mathrel{-\!\!*} \forall \sigma.\, \mathsf{SI}(\sigma) \;{}_{\mathcal{E}}\!\Rrightarrow\!\!\!\!\!\!\;\;{}_{\emptyset}\; \mathsf{Progress}(e, \sigma) * \mathsf{Preservation}(e, \sigma)$$

Progress The program e must be able to take one more thread-local step forward:

$$\mathsf{Progress}(e, \sigma) \triangleq \exists e', \sigma'. \langle e, \sigma \rangle \to \langle e', \sigma' \rangle$$

For instance, when e is about to load from a location, it must be demonstrated that the corresponding cell is allocated within the heap.

Preservation The abstract view and the physical heap must remain consistently linked throughout the execution of e:

$$\mathsf{Preservation}(e, \sigma) \triangleq \forall e', \sigma'. \langle e, \sigma \rangle \to \langle e', \sigma' \rangle * \triangleright\, {}_{\emptyset}\Rightarrow_{\mathcal{E}} \mathsf{SI}(\sigma') * \mathsf{wp}_{\mathcal{E}}\, e'\, \{Q\}$$

This preservation is established recursively: for a single reduction step from σ to σ', the state interpretation is updated from $\mathsf{SI}(\sigma)$ to $\mathsf{SI}(\sigma')$; and the weakest precondition of e' enforces this link for the reminder of the execution. The use of modalities $\triangleright\, {}_{\emptyset}\Rightarrow_{\mathcal{E}}$ is for soundness. The fancy update modality restores the invariant mask to $\mathcal{E}$ to ensure that invariants opened for evaluating e are reestablished after the atomic step. This mechanism effectively ensures that invariant resources are only available temporarily for atomic steps, as in Iris-ht-inv. The recursive occurrence of the weakest precondition is placed behind a later modality to guarantee the existence of a guarded fixed point for its definition. The new state interpretation is also behind the later modality so that the one in front of the invariant resources (Iris-inv-open) can be eliminated after the step.

Soundness and Adequacy

With an instantiated weakest precondition, one can define a tailored Iris program logic for the language by formulating proof rules for its primitives using Hoare triples. Such a logic must be proven consistent and useful. Consistency requires the *soundness* of the proof rules: the proof rules must be sound with respect to the instantiated weakest precondition and the ghost theory; usefulness requires the *adequacy* of the logic: one should be able to extract the verification results with respect to the semantics of programs from the Iris program logic to the meta-logic.

Soundness of proof rules To show the soundness of the proof rules for the toy language, we need a ghost theory for its heap. The ghost theory for heaps σ consists of the full view $\mathsf{A}(\sigma) \triangleq \boxed{\bullet\sigma}^{\gamma}$ and the fragmental view $\mathsf{F}(\sigma) \triangleq \boxed{\circ\sigma}^{\gamma}$, both implemented using the Authoritative CMRA. The following rules on the two views can be derived:

GT-heap-agree
$$\mathsf{A}(\sigma) * \mathsf{F}(\sigma') \vdash \sigma' \subseteq \sigma$$

GT-heap-update
$$\frac{l \in \mathsf{dom}(h')}{\mathsf{A}(\sigma) * \mathsf{F}(\sigma') \Rrightarrow \mathsf{A}(\sigma[l \mapsto v]) * \mathsf{F}(\sigma'[l \mapsto v])}$$

GT-heap-alloc
$$\mathsf{A}(\sigma) \Rrightarrow \exists l \notin \mathsf{dom}(\sigma). \mathsf{A}(\sigma[l \mapsto v]) * \mathsf{F}(\{l \mapsto v\})$$

The full view serves as the abstract heap state in the state interpretation $\mathsf{SI}(\sigma) \triangleq \mathsf{A}(\sigma)$, while the fragmental view implements the points-to assertion $x \hookrightarrow v \triangleq \mathsf{F}(\{x \mapsto v\})$.

This ghost theory suffices to establish the rules for allocation, read, and write with respect to the definition of the weakest precondition and the operational semantics:

Iris-ht-load
$\{x \hookrightarrow v\}\ !x\ \{w.\, w = v * x \hookrightarrow v\}$

Iris-ht-store
$\{x \hookrightarrow v\}\ x := v'\ \{x \hookrightarrow v'\}$

Iris-ht-alloc
$\{\mathsf{True}\}\ \mathsf{ref}(v)\ \{x.\, x \hookrightarrow v\}$

Iris-ht-load We use GT-heap-agree to conclude that $\sigma[x] = v$, as required by Opsem-tl-load.

Iris-ht-store We use GT-heap-update to update the points-to assertion when updating the heap from σ to $\sigma[x \mapsto v']$ according to Opsem-tl-store.

Iris-ht-alloc We use GT-heap-alloc to allocate a new points-to assertion when extending the heap from σ to $\sigma[x \mapsto v]$ for a fresh location x according to Opsem-tl-alloc.

Adequacy The adequacy theorem allows us to lift verification results from the program logic to the meta-logic. This removes the entire program logic construction and the Iris base logic presented in this section from the trusted computing base, relying only on the meta-logic −− in this case, the Coq/Rocq proof assistant.

The adequacy theorem can be formulated in different ways to reflect varying strengths. Theorem 2.2.2 presents a simple adequacy theorem for our toy language.

Theorem 2.2.2 (Adequacy theorem)**.** *Given an initial heap σ, programs $\vec{e}$, meta-level propositions $\vec{\varphi}$ (one for each thread), ff we have a thread-pool reduction trace $\langle \vec{e}, \sigma\rangle \to^*_{\mathsf{tp}} \langle \vec{e}', \sigma'\rangle$, and want to show that each φ holds for the corresponding thread if the thread terminates in the meta-logic, it suffices to show the following in the Iris logic:*

$$\vdash \Rrightarrow_{\top} \exists \mathsf{SI}.\, \mathsf{SI}(\sigma) * \mathop{\Large *}_{e,\varphi \in \vec{e};\vec{\varphi}} \mathsf{wp}_{\top}\, e\, \{\lceil \varphi \rceil\}$$

This entailment requires instantiating the state interpretation SI, allocating it for the initial heap, and performing verification using weakest preconditions (with $\top$ mask including all namespaces) for all threads using the meta-level propositions as the postconditions.

The proof of this theorem requires induction on the reduction trace, unfolding the weakest preconditions along the way, and relying on the soundness of the Iris base logic (Theorem 2.2.1).

2.3 Relaxed Memory Concurrency

The performance of modern multi-core programs benefits substantially from hardware optimisations such as caching and speculation. Since the introduction of

multi-core processors, there have been significant advancements in their performance. One such example is the introduction of multi-layered caching between CPU cores and memory to reduce communication latency. These caches prioritise speed over strict memory access order, significantly enhancing performance by reducing dependence on slower memory accesses. However, such optimisations lead to out-of-order execution of instructions, where normal memory accesses may not occur in the order specified by the program. To compensate this side effect, concurrent code has to use a family of memory barriers and stronger memory accesses to ensure ordering at various granularities.

The sequential consistency (SC) model [Lamport, 1979] is a widely adopted concurrency model for study of concurrent programs. It requires that in a valid execution, the effects of all concurrent memory operations are ordered sequentially, and the sequential order respects the program order of the threads. A substantial body of work on concurrent program verification is based on this model. However, SC is *not* a correct modelling of real-world concurrency as it disallows out-of-order behaviours. Instead, the concurrency of real-world hardware and programming languages is modelled using *relaxed* memory models.

High-level languages and hardware Both high-level languages and hardware architectures define their own memory models. For hardware, memory models are part of the architectural specifications that formally define the observable behaviours of processors as abstractions over their complex implementations.

Defining relaxed memory models for high-level languages is more challenging, as they must *also* account for compilation schemes targeting different architectures. Designing these models requires handling different relaxed hardware architectures and accommodating compiler optimisations that may restructure programs by re-ordering, removing, or merging memory operations.

Litmus testing Litmus testing is a method for understanding intricate relaxed concurrency behaviours. Litmus tests are tiny concurrent programs that capture specific memory access patterns. Their execution outcomes reflect the characteristics of relaxed memory models which can be simulated using tools such as herd [Alglave et al., 2014]. We present litmus tests in a simple assembly-like language: str $[x]$ v denotes a memory store at address x with value v; $r :=$ ldr $[y]$ denotes a memory load from address y into register r. For instance, message passing (MP) captures

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

Figure 2.2: MP in SC: $r_1 = 1 \land r_2 = 42$. The initial value of the two locations are zero. The symbol before each instruction is the label of the instruction.

the pattern of one thread passing a message to the other thread by writing the message to a shared location *data*, and uses another location *flag* for notifying the

reader thread the completion of the write, as depicted in Figure 2.2. In sequential consistency concurrency, one can expect the reader thread to only read the payload 42 from *data* if it reads 1 from *flag*. This is because the two writes in the writer thread are executed in program order, which ensures that the write to *data* happens before the write to *flag*. Under sequential consistency, the reader thread thus can observe the two writes only in the same order, and the two reads are also executed in program order. As we will see later, this is however not always the case in relaxed concurrency because of the reordering of instructions.

Structure In this preliminary section, we give an overview on relaxed memory concurrency with definitions and litmus tests. The section is structured as follows:

Section 2.3.1 describes two styles of formalising memory models, using two formalisations of (a subset of) x86-TSO as examples.

Section 2.3.2 discusses two relaxed memory models: C/C++ and Arm-A.

Section 2.3.2 highlights verification work on relaxed memory concurrency.

2.3.1 Two Styles of Formal Semantics

There are two widely used styles for formally specifying concurrency models: operational and axiomatic.

Operational An operational concurrency model is an *abstract machine* stimulating the concurrency behaviours. More specifically, an abstract machine is a labeled transition system (LTS), whose valid transitions are allowed execution outcomes. The states of the LTS include information about the program state; and the transitions of LTS update these information, modelling a small execution step. They often have a microarchitectual flavour, abstracting behaviours of processor-internal hardware units and their interactions.

Operational models are executable: one can implement models such that they execute incrementally, allowing inspecting the relaxed behaviours interactively with e.g. the RMEM tool [Sarkar et al., 2024]; they are also relatively intuitive: it is usually easier to get intuition on certain relaxed behaviours by considering how the abstract machine would run.

Axiomatic An axiomatic model is a *predicate* over candidate executions. Candidate executions are all possible well-formed executions of a program, including those that are inconsistent with the model. A candidate execution consists of memory events and various relations between the events. The axiomatic model defines which of those candidate executions are consistent, often by posing constraints on the relations.

Axiomatic models are more abstract than their operational counterparts, thus providing less intuition; they are not trivially executable due to their non-operational nature. Isla [Armstrong et al., 2021] makes axiomatic models executable by translating the predicates to SMT constraints.

Ideally, one has complementary, equivalent operational and axiomatic models for the same concurrency system. In the rest of this subsection, we present the x86-TSO (TSO for short) memory model in the two styles, and illustrate how the model accepts or rejects certain behaviours of representative litmus tests. For brevity, we do not consider read-modify-write operations and memory barriers.

Operational

The TSO model considers that each processor has a *write buffer* that connects to the shared memory. The buffer holds all pending writes of the processor, and works like a queue: it pops writes to the shared memory in first-in-first-out order. A processor reading from a given reads from the latest relevant entry of its own write buffer if there is one, and shared memory otherwise.

We define the state of the abstract TSO machine σ to track the state of all write buffers and the shared memory:

$$\sigma \triangleq \langle buf, mem \rangle$$

where buf maps from thread identifiers to the private write buffers of the threads, and mem maps from memory addresses to values.

LTS Formally, we define the abstract TSO machine as an LTS, with the following transition on machine states:

$$\sigma \xrightarrow{a:\alpha} \sigma'$$

where σ and σ' are the machine states before and after an execution step, α is the memory action emitted from the machine step, and $a = \langle i, t \rangle$ is a pair of an action identifier i and the thread identifier t of the thread performing the action. A memory action α can be

Write x v A write action with address x and value v.

Read x v A read action with address x and value v.

Dequeue a x v A dequeue action with address x and value v, which moves the write a from the write buffer to the shared memory.

The machine can make the following transitions according to the action:

$$\textsc{TSO-op-read-mem}\ \frac{\sigma.mem[x] = v \qquad \mathsf{NotIn}(\sigma.buf[a.t], x)}{\sigma \xrightarrow{a:\mathsf{Read}(x,v)} \sigma} \qquad \textsc{TSO-op-read-buf}\ \frac{\mathsf{IsLatest}(\sigma.buf, a':\mathsf{Write}(x,v))}{\sigma \xrightarrow{a:\mathsf{Read}(x,v)} \sigma}$$

$$\textsc{TSO-op-write}\ \frac{buf' = \sigma.buf[a.t \mapsto \mathsf{enqueue}(\sigma.buf[a.t], a:\mathsf{Write}(x,v))]}{\sigma \xrightarrow{a:\mathsf{Write}(x,v)} \langle buf', \sigma.mem\rangle}$$

$$\textsc{TSO-op-dequeue}\ \frac{a':\mathsf{Write}(x,v) = \mathsf{top}(\sigma.buf[a.t]) \qquad buf' = \sigma.buf[a.t \mapsto \mathsf{dequeue}(\sigma.buf[a.t])]}{\sigma \xrightarrow{a:\mathsf{Dequeue}(a',x,v)} \langle buf', \sigma.mem[x \mapsto v]\rangle}$$

TSO-op-read-mem This rule allows the thread $a.t$ to read x from memory when there is no write at x in its write buffer.

TSO-op-read-buf This rule allows the thread $a.t$ to read from the latest write at x in its write buffer.

TSO-op-write This rule allows the thread $a.t$ to buffer write a.

TSO-op-dequeue This rule allows the thread $a.t$ to move the oldest write of its buffer to the memory.

Store buffering To justify outcomes of litmus tests using the operational model, one needs to find a transition of the LTS from the initial state σ_0 where all write buffers are empty and all addresses are initialised with value 0 in the memory. The store buffering (SB) litmus test in Figure 2.3 captures that in TSO local write-read pairs might be reordered.

$$\begin{array}{l|\!|l} a\colon \mathsf{str}\ [y]\ 1 & c\colon \mathsf{str}\ [x]\ 1 \\ b\colon r_1 := \mathsf{ldr}\ [x] & d\colon r_2 := \mathsf{ldr}\ [y] \end{array}$$

Figure 2.3: SB in TSO: $r_1 = 0 \wedge r_2 = 0$ allowed. The initial value of the two locations are zero.

We can use the operational model to explain why the outcome $r_1 = 0 \wedge r_2 = 0$ is allowed. This is because two threads can first push their writes to their write buffers, and read the value at the other location before the buffered writes are dequeued to the memory, as in the following sequence of actions:

$$\langle a, L\rangle{:}\mathsf{Write}(y,1)\ ;\ \langle c, R\rangle{:}\mathsf{Write}(x,1)\ ;\ \langle b, L\rangle{:}\mathsf{Read}(x,0)\ ;\ \langle d, R\rangle{:}\mathsf{Read}(y,0)\ ;$$
$$\langle \_, L\rangle{:}\mathsf{Dequeue}(\langle a, L\rangle, y, 1)\ ;\ \langle \_, R\rangle{:}\mathsf{Dequeue}(\langle c, R\rangle, x, 1)$$

where we assign the identifier L to the left thread and R to the right thread.

Axiomatic

A candidate execution of the TSO model consists of a candidate pre-execution $\langle E, \mathrm{po}\rangle$, and a witness $\langle \mathrm{rf}, \mathrm{co}\rangle$, where

$\underline{E}$ is a set of memory events each labeled with an unique identifier. Each memory operation of the program may emit one or more memory event, each representing some effect of the memory operation. For TSO, the events are $\mathsf{W}\ x\ v$ for write operation str $[x]\ v$, and $\mathsf{R}\ x\ v$ for the read operation that reads value v. The event identifier (*Eid*) also carries the thread ID of the thread that the event belongs to, to distinguish if two events belong to the same thread.

$\underline{\mathrm{po}}$ is the *program order* relation for events ($\mathit{Eid} \times \mathit{Eid}$), reflecting the syntactic order of memory operations. This relation is irreflexive and transitive, and only relates events from the same thread.

$\underline{\mathrm{rf}}$ is the *read-from* relation that relates each write to all reads that read from it. One read event can only be related to one write event, and the two events must have the same address and value. Reads can also read from initial writes with default value 0. Initial writes to the memory addresses used in the program are modelled as write events that do not correspond to explicit memory operations.

$\underline{\mathrm{co}}$ is the *coherence* relation that relates writes of the same address. This relation is irreflexive, transitive, and total for writes of the same address.

Auxiliary relations and relational algebra We use notations $e\ \mathcal{R}\ e'$ or $(e, e') \in \mathcal{R}$ for two events with *Eid*s e and e' related by relation $\mathcal{R}$. Since many relations capture ordering between events, we often say e is $\mathcal{R}$-ordered after e' for $e\ \mathcal{R}\ e$. There are also some auxiliary relations:

$\underline{\mathrm{loc}}$ is the same-location relation that relates events at the same location.

$\underline{\mathrm{int}}$ is the internal relation that relates events from the same thread.

$\underline{\mathrm{ext}}$ is the external relation that relates events from different threads.

The consistency condition of a memory model often refers to compound relations defined using the base relations of a candidate execution. For convenience, we use relation algebra to compose relations. For relations $\mathcal{R}, \mathcal{R}'$ and event set S:

$\underline{\text{Union}}\ \ \mathcal{R} \mid \mathcal{R}'$ or $\mathcal{R} \cup \mathcal{R}' \triangleq \{(e, e') \mid (e, e') \in \mathcal{R} \vee (e, e') \in \mathcal{R}'\}$

$\underline{\text{Intersection}}\ \ \mathcal{R}\ \&\ \mathcal{R}'$ or $\mathcal{R} \cap \mathcal{R}' \triangleq \{(e, e') \mid (e, e') \in \mathcal{R} \wedge (e, e') \in \mathcal{R}'\}$

$\underline{\text{Sequential composition}}\ \ \mathcal{R}\,;\mathcal{R}' \triangleq \{(e, e') \mid \exists e''.\ (e, e'') \in \mathcal{R} \wedge (e'', e') \in \mathcal{R}'\}$

$\underline{\text{Minus}}\ \ \mathcal{R} \setminus \mathcal{R}' \triangleq \{(e, e') \mid (e, e') \in \mathcal{R} \wedge (e, e') \notin \mathcal{R}'\}$

$\underline{\text{Inverse}}\ \ \mathcal{R}^{-1} \triangleq \{(e, e') \mid (e', e) \in \mathcal{R}\}$

$\underline{\text{Identity}}\ \ [S] \triangleq \{(e, e) \mid e \in S\}$

We then define a few more auxiliary relations using relational algebra:

$\mathsf{fr} \triangleq \mathsf{rf}^{-1}; \mathsf{co}$ is the from-read relation that relates each read with the writes that are co-after the write that paired with the read with rf.

$\mathsf{rfe} \triangleq \mathsf{rf} \cap \mathsf{ext}$ is the external fragment of rf. Likewise, we have coe, fre.

$\mathsf{rfi} \triangleq \mathsf{rf} \cap \mathsf{int}$ is the internal fragment of rf. Likewise, we have coi, fri.

$[\mathsf{W}]$ is the identity relation for all write events.

$[\mathsf{R}]$ is the identity relation for all read events.

The model We are now ready to give the TSO axiomatic model. We say a TSO candidate execution is a consistent TSO execution if it satisfies the following two conditions:

SC-per-location $\mathsf{acyclic}(\mathsf{po\text{-}loc} \cup \mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr})$, where $\mathsf{po\text{-}loc} \triangleq \mathsf{po} \cap \mathsf{loc}$. This condition requires that all memory access at the same location have sequentially consistent behaviours by requiring acyclicity of the union of the coherence orders between events at same address. Coherence is a fundamental property that most hardware architectures and many programming languages, but not all distributed systems, enforce.

External-visibility $\mathsf{acyclic}((\mathsf{po} \setminus ([\mathsf{W}]; \mathsf{po}; [\mathsf{R}])) \cup \mathsf{rfe} \cup \mathsf{coe} \cup \mathsf{fre})$. We refer to the union of the four orders as ob (ordered-before), which is the order that captures the synchronisation between threads. The ob relation is externally visible, meaning that if two events are ordered by it, then this ordering is visible to all threads.

Litmus tests We now illustrate the TSO axiomatic model with several litmus tests.

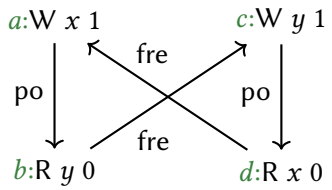
SB TSO allows the SB outcome in Figure 2.3. As shown in Figure 2.4a, the execution is allowed since no forbidden cycles are formed. In particular, since $[\mathsf{W}]; \mathsf{po}; [\mathsf{R}]$ is excluded from the synchronisation order, a and b, c and d are not ob ordered.

MP TSO disallows the $r_1 = 1 \wedge r_2 = 0$ outcome of MP (Figure 2.2). Execution in Figure 2.4b is forbidden because of the ob cycle: a po b rfe c po d fre a.

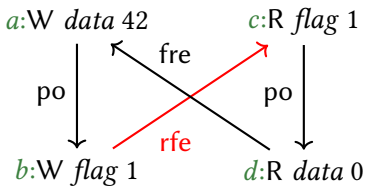
Other styles

The promising semantics [Kang et al., 2017; Lee et al., 2020; Pulte et al., 2019] is proposed to prevent the out-of-thin-air problem of C11. It is a timestamp-based operational model with a promising step that allows threads to promise to perform a write in the future, and requires a certification that ensures the promises can indeed be implemented.

Jeffrey et al. [2022] propose a denotational semantics that supports sequential composition as an alternative proposal for fixing the out-of-thin-air problem.



(a) An allowed execution of SB. b and c are related by fre because b reads from the initial write of y, which is co-ordered before c. Likewise, d and a are related by rfe.



(b) The forbidden execution of MP. d and a are related by fre because d reads from the initial write of *data*, which is co-ordered before a.

Figure 2.4: Illustration on the TSO axiomatic model with litmus tests

2.3.2 C/C++11 and Arm-A

In this subsection we provide a preliminary introduction to the C/C++11 and the (user) Arm-A memory model, as they will be discussed in the publications of this dissertations.

C/C++11 The C/C++11 memory model [Batty et al., 2011; Boehm and Adve, 2008] (C11 hereafter) is a high-level language model that supports various low-level access modes for different access strengths. It classifies memory accesses into *non-atomic* accesses and *atomic* accesses.

Non-atomic accesses are not allowed to have races: any race on non-atomic accesses results in undefined behaviour (UB) which gives no execution guarantees for the program.

Atomic accesses can be annotated with different access modes and are allowed to have races. Examples of the modes include `memory_order_release` for stores that creates ordering between the annotated store and all program-order-before accesses; `memory_order_acquire` for loads that creates ordering between the annotated load with all program-order-after accesses.

As a language-level model, C11 is sound with respect to the compiler optimisations, but is excessively weak since it suffers from the out-of-thin-air problem: the model allows certain undesirable execution outcomes that the language implementations should not allow. This problem is inherited by other language-level models based on C11, for instance Rust. There have been multiple proposals for fixing the 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 none has been accepted officially. Among these, Repaired C11 (RC11) [Lahav et al., 2017] is a relatively well adapted fix-up in research. It eliminates the out-of-thin-air problem by significantly strengthening C11, to rule out the read-write reordering of relaxed accesses. For instance, the weak outcome of Load Buffering (LB) in Figure 2.5 is forbidden by RC11, which is allowed by very relaxed hardware models like Arm-A.

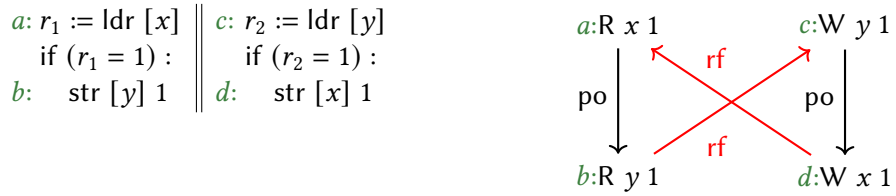


Figure 2.5: LB in RC11: $r_1 = 1 \wedge r_2 = 1$ forbidden. This execution has a forbidden cycle of po $\cup$ rf.

Arm-A Arm-A [Arm Ltd., 2023] is Arm's application profile architecture, which has a *very relaxed* memory model that is much weaker than x86-TSO: Arm-A instructions are executed out-of-order *by default* unless violating coherence or explicit ordering. User Armv8-A is a relatively well-studied memory model for the eighth version of the Arm-A profile. It is a model that is slightly stronger but much simpler than the models for previous versions, and focuses on only the user-land features: no instruction-fetch or any system features. We just use Arm-A to refer to this model from now on for convenience. Arm-A has a microarchitecture-flavoured operational model and an axiomatic model [Alglave et al., 2021; Deacon, 2016; Pulte et al., 2018] that are proven equivalent. We elaborate on the basic of the axiomatic formulation since it is used to develop this dissertation.

Similar to TSO, Arm has a standard SC-per-location requirement and an external visibility requirement that requires acyclicity of its synchronisation order ob. The definition of ob however diverges. We do not present the full ob definition here since it will be discussed later in the publication part. Instead, we highlight following elements of the model that can form ob:

Dependencies Being an architecture memory model, Arm-A has syntactic dependencies which can enforce synchronisation:

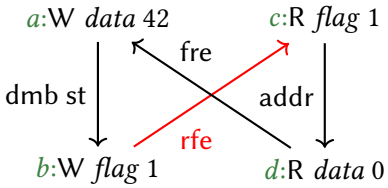
- addr is address dependency: $(e, e') \in$ addr if there is a data flow from the value of e to e''s address.
- data is data dependency: $(e, e') \in$ data if there is a data flow from the value of e to the value of write e'.
- ctrl is control dependency: $(e, e') \in$ ctrl if the execution of e' depends on the value of e.

Barriers Arm-A has a family of data memory barriers dmb that enforce synchronisation:

- dmb st is a barrier that orders stores: two same-thread events are ordered by ob if there is a dmb st in between.
- dmb ld is a barrier that orders loads and any event: a load and any event are ordered if there is a dmb ld in between.
- dmb sy is a full barrier: two events are ordered if there is a dmb sy in between.

$$\begin{array}{l|l} a\text{: str } [data]\ 42 & c\text{: } r_1 := \text{ldr } [flag] \\ \quad \text{dmb st} & d\text{: } r_2 := \text{ldr } [data + r_1 - r_1] \\ b\text{: str } [flag]\ 1 & \end{array}$$

(a) $r_1 = 1 \wedge r_2 = 0$ forbidden.



(b) The forbidden execution of MP+dmbst+addr. The four depicted edges form an ob cycle. a and b are two writes thus ob-ordered by dmb st. c and d are ob-ordered because of the address dependency: The address of d depends on the value of c.

Figure 2.6: MP+dmbst+addr in Arm-A

Due to its highly concurrent nature, Arm-A allows the undesired $r_1 = 1 \wedge r_2 = 0$ outcome for the plain MP test of Figure 2.2. To forbid this outcome, one can leverage dependencies and barriers, as shown in the MP variant of Figure 2.6a. A key characteristic of the very relaxed aspect of Arm-A is that it permits read-write re-ordering, allowing the LB outcome of Figure 2.5.

Verification

Given its semantic foundation, formal verification for relaxed memory concurrency has been an active research topic. Two mainstream approaches exist: deduction-based verification using program logics, and enumeration-based verification using model checking. We briefly discuss related work in these approaches below.

Model checking For model checking [Clarke, 1997] against relaxed memory models, one key challenge is how to enumerate all executions (or interleavings) efficiently, with minimal memory footprints and time overhead. The combination of concurrency and out-of-order execution results in a vast state space, often exponential in the size of the program.

Stateless model checking (SMC) [Godefroid et al., 1996] is a model checking technique that does not need to store visited states. SMC has been applied to implement fast checking of numerous memory models, including SC, fragments of C11, RC11, TSO, POWER, Arm-A [Abdulla et al., 2015, 2019, 2016, 2018; Kokologiannakis et al., 2018, 2022]. To mitigate state-space explosion, SMC often employs partial-order reduction (POR) and its refined version, Dynamic POR (DPOR) [Flanagan and Godefroid, 2005]. These algorithms define an equivalence on executions and ensure that at least one representative execution in each equivalence class is checked while guaranteeing full exploration of possible behaviours. Further refinements to DPOR have

been proposed to enhance both it efficiency and capability. An optimal revision of DPOR [Abdulla et al., 2018], applied to the release-acquire fragment of C11, explores exactly one execution per equivalence class. TruSt model checker [Kokologiannakis et al., 2022, 2023] implements advanced DPOR algorithms for RC11 and achieves provable optimality in both time and space (polynomial memory consumption). GenMC [Kokologiannakis et al., 2019; Kokologiannakis and Vafeiadis, 2020, 2021; Marmanis et al., 2025] is a model checker featuring multiple DPOR enhancements, and is parameterised over various memory models that satisfy certain conditions. Other SMC-based model checkers include CDSChecker [Norris and Demsky, 2013] (for C11) and Nidhugg [Abdulla et al., 2015, 2019, 2018].

Additionally, Dartagnan [de León et al., 2020] is an SMT-based relaxed memory model checker employing bounded model checking (BMC). It translates verification problems into satisfiability problems and delegates them to SMT solvers.

Program logic We highlight key developments in separation logic and refer readers to the publications for extensive discussions. Most separation logics for relaxed memory concurrency target high-level language memory models.

Starting from RSL [Vafeiadis and Narayan, 2013], there is a long line of work [Dang et al., 2020; Doko and Vafeiadis, 2016, 2017; He et al., 2016; Kaiser et al., 2017; Turon et al., 2014] on building separation logic for the C11 memory model. Overall, these logics trade off the expressive power of the logic and the coverage of the memory model. RSL [Vafeiadis and Narayan, 2013] supports a substantial subset of C11 but limits or forbids resource transfer for specific accesses. FSL [Doko and Vafeiadis, 2016] extends RSL to support memory fences and allow constrained resource transfer via relaxed accesses, but lacks ghost resources. FSL++ [Doko and Vafeiadis, 2017] incorporates ghost resources to FSL, but is sound only for the strengthened RC11 model. GPS [Turon et al., 2014] is based on a (stronger) subset of C11 and supports ghost resources. All the aforementioned logics rely on axiomatic memory models, with soundness proofs derived from intricate semantic models built from scratch. iGPS [Kaiser et al., 2017] ports GPS to Iris by instantiating Iris with an operational reformulation of the memory model of GPS, which demonstrates the benefits of building logics using Iris. iRC11 [Dang et al., 2020] extends iGPS by integrating the ideas of FSL++ and builds upon an operational reformulation of RC11.

Cosmo [Mével et al., 2020] is an Iris-based logic targeting the memory model of multicore OCaml, which is stronger than that of C11.

2.4 Further Related Work

In this section, we discuss efforts on using separation logic for modular verification of low-level and/or realistic systems. We in particular focus on those that are not discussed in the papers of Part II.

2.4.1 Realistic Modelling

This subsection discusses related work on using separation logic to reason about programs with more realistic models. These works model realistic hardware and/or language behaviours beyond those of idealised language semantics.

Crash-safety and file systems

Perennial [Chajed et al., 2019] is an Iris-based framework for reasoning about concurrent and *crash-safe* programs. Concurrent programs may experience crashes at any time during execution and rely on a recovery program to maintain system consistency. The paper models crashes in concurrent settings and focuses on verifying file storage systems implemented in the Go language. They implement the Goose translator to translate a subset of Go to Coq/Rocq definitions, on which they employ Perennial for verification. Perennial extends the Iris framework with new techniques for reasoning about concurrent crash-safety in a thread-modular way. Perennials marks resources and Hoare triples with versions to indicate crash occurrences, which enables reasoning about concurrent programs and their recovery programs separately. Perennial is evaluated against a small crash-safe mail server. The evaluation shows that the verified server has better performance than a previously verified variation, and requires less verification effort.

GoJournal [Chajed et al., 2021] is a concurrent and crash-safe journaling system verified using Perennial 2.0. Perennial 2.0 includes substantial improvements over the initial version in being able to specify and reason about GoJournal in a modular way. This is to capture the crash atomicity essence of GoJournal's top-level operations: the clients of GoJournal should use these operations as if they were atomic, and without thinking about crashes.

Non-volatile memory

Spirea [Vindum and Birkedal, 2023] is an Iris-based separation logic for non-volatile memory. Non-volatile memory combines the advantages of volatile memory and durable storage: it offers fast random access and keeps its data resilient to machine crashes. Memory models for non-volatile memory are both weak and persistent. In weak and persistent memory, crashes are non-deterministic: the program state after a crash is not deterministically determined by the program state right before the crash. This work presents the first CSL that accounts for this, taking inspirations from prior separation logics for weak or persistent memory. Specifically, this work generalises the Perennial logic for persistency and integrates ideas from the line of work on relaxed memory logics for weak concurrency. The logic benefits significantly from Iris's advanced separation logic features, which are crucial for modular verification — something absent in previous non-separation logics for persistent memory [Bila et al., 2022; Raad et al., 2020]. Using its novel crash-aware invariants and a collection of modalities, Spirea provides high-level reasoning rules and supports thread-local reasoning and sound resource transfer for weak persistency. It is evaluated by

verifying various examples against high-level abstract specifications. In particular, it is demonstrated by verifying concurrent durable data structures with null-recovery: concurrent data structures whose data consistency is resilient to non-deterministic crashes without needing any recovery.

2.4.2 Realistic Languages

In addition to reasoning about idealised toy languages with realistic modelling, separation logic has been applied to reason about substantial portions of semantics of realistic programming languages. This subsection focuses on these work.

WebAssembly

Iris-Wasm [Rao et al., 2023] is a program logic for the WebAssembly (Wasm) 1.0 language specification. Wasm is a low-level stack language with an expressive module system. One key property of the language is that it provides coarse-grained encapsulation to modules: the memory of modules cannot be accessed by other modules unless the module exports its whole memory explicitly. To compose them, Wasm modules need to be instantiated by the host. Based on Iris, the program logic achieves compositional higher-order reasoning for Wasm modules: it can verify modules individually, and compose them when the host instantiates the modules. Another contribution of Iris-Wasm is that it captures module encapsulation with a notion of robust safety, which allows a verified module to be combined with arbitrary unknown modules. This is achieved by defining a logical relation on top of the program logic. This work makes an impressive contribution of applying Iris to a full industrial language semantics. Iris-MSWasm [Legoupil et al., 2024] extends Iris-Wasm, for the MSWasm proposal that extends Wasm with finer-grained encapsulation.

C

The Verified Software Toolchain (VST) [Appel, 2012] contains a separation logic targeting the C programming language. The logic is based on and proven sound with respect to the semantics of CompCert C light, a formalisation of a large subset of C99 used by the CompCert project. It supports an extensive collection of language features, for instance function pointers, and recursive definitions, so that it can be used to verify real-world C programs. Since the logic shares the C semantics with CompCert, C programs verified in VST can be compiled to correct semantic-preserving assembly code by the verified CompCert compiler.

VST 3.0 [Mansky and Du, 2024] port the VST logic to Iris, by instantiating Iris with the CompCert C semantics, and recover the reasoning rules of VST in the new Iris-based logic. By doing so, the new logic enjoys the benefits of both VST and Iris: it obtains the same expressive power and soundness results of VST and better abstractions and extensibility provided by the advanced logical features and

soundness of Iris. This work includes substantial efforts on adapting Iris to solve the mismatch between the default semantics shape required by Iris and the complex CompCert C semantics, and implement the VST logical features and automation missing in Iris. This work an impressive example showing how a modern higher-order separation logic like Iris can scale to large language semantics.

Rust

RustBelt [Jung et al., 2018a] is a project on formalising and reasoning about Rust programs. It formalises a core fragment of Rust and its type system, and uses Iris to build a lifetime logic that captures borrowing, a key idea of the Rust type system. It establishes semantic type soundness for the formalised type system using its lifetime logic. Semantic typing can show safety of programs that are not syntactically well typed. For Rust, semantic typing is used to show that the Rust libraries implemented using unsafe features only expose a type safe interface to clients written in safe Rust, such that safety for the composition can be obtained.

RustHornBelt [Matsushita et al., 2022] extends RustBelt for verifying functional correctness.

2.4.3 Mechanisation of Separation Logic

There have been efforts to mechanise separation logic in proof assistants so that soundness proofs, which often involve complicated semantic models, can be machine-checked. Even better, some mechanisations also offer users the opportunity to conduct interactive and foundational verification using the logic.

FCSL [Nanevski et al., 2014] is a concurrent separation logic mechanised in Coq/Rocq [Sergey et al., 2015a]. The logic is embedded in Coq/Rocq as a DSL, and is also proven sound with respect to its semantic model in Coq/Rocq. It is used to verify several fine-grained concurrent modules.

Iris (including its semantic model and soundness) is mechanised in Coq/Rocq. Iris Proof Mode (IPM) [Krebbers et al., 2017b] is the interactive proof framework for Iris implemented as an extension to Coq/Rocq, which provides tactics for manipulating Iris proof terms, offering a similar experience as doing proofs in plain Coq/Rocq. IPM is one of the key factor contributing to the success of Iris. Most of the Iris-based work thereafter are mechanised. MoSeL [Krebbers et al., 2018] generalises IPM to make it applicable to both affine and linear separation logics.

The VST [Appel, 2012] logic is also mechanised, with distinct mechanisation infrastructure and architecture from Iris. Its latest development, VST 3.0 [Mansky and Du, 2024], however is based on Iris and instantiates IPM.

3

Results

This dissertation contains text and material from the following publications and manuscripts based on the work conducted during my Ph.D:

[Liu et al., 2023b] Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023b. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1438–1462. doi:10.1145/3591279

[Hammond et al., 2024] 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 (2024), 604–637. doi:10.1145/3632863

[Liu et al., 2024] 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. (2024), In submission

[Liu et al., 2025] Zongyuan Liu, Lars Birkedal, and Jean Pichon-Pharabod. 2025. First Steps Towards AxSL+. (2025)

The following article includes work conducted during my Ph.D but is not part of this dissertation:

[Pérami et al., 2024] Thibaut Pérami, Zongyuan Liu, Nils Lauermann, Brian Campbell, Alasdair Armstrong, Thomas Bauereiss, and Peter Sewell. 2024. Reusable Rocq semantics of modern relaxed architectures. (2024), In submission

3.1 Coq/Rocq Mechanisations

A significant portion of the work presented in this dissertation is foundational. Given the complexity arising from detailed modelling and intricate logical abstractions,

mechanisation is the preferred method for ensuring the correctness of the results. The following repositories contain the respective Coq/Rocq mechanisations:

`https://github.com/logsem/VMSL` is the repository that contains the mechanisation of the results of [Liu et al., 2023b].

`https://github.com/logsem/AxSL` is the repository that contains the mechanisation of the results of [Hammond et al., 2024] and several major results of [Liu et al., 2024].

`https://github.com/rems-project/archsem` is the repository that contains the mechanisation of the results of [Pérami et al., 2024].

3.2 Personal Contributions

My contributions to each of the aforementioned works are as follows:

[Liu et al., 2023b] I am the lead author of this work. I led the research and the development of the mechanisation. The mechanisation also includes contributions from Sergei Stepanenko and Amin Timany. I wrote the majority of the paper, with editorial input from Lars Birkedal and Jean Pichon-Pharabod, and feedback from all co-authors.

[Hammond et al., 2024] I am the co-lead author of this work. I led the research together with Angus Hammond, and mechanised the majority of the results of the work in Coq/Rocq. I authored Section 4 to 6 (except for 5.5) with editing help and feedback from the co-authors, and contributed to writing and editing the remaining sections, led by Jean Pichon-Pharabod and Peter Sewell.

[Liu et al., 2024] I am the lead author of this work. This work is an extension of [Hammond et al., 2024], including generalisation to the original work, elaborated examples, and significantly refined presentation. I led both research and writing, with editorial feedback from Lars Birkedal and Jean Pichon-Pharabod.

[Pérami et al., 2024] I am an author of this work. I contributed to the early development of its Coq/Rocq mechanisation, assisting in mechanising definitions, refining tactics, and proving lemmas required for [Hammond et al., 2024]. I also contributed to drafting the mechanisation of one example.

[Liu et al., 2025] I am the lead author of this work, for both research and writing. The writing process received editing help from Jean Pichon-Pharabod.

3.3 Structure

The following list outlines the correspondence between the included papers and the content of the individual chapters in Part II:

Chapter 4 contains the content of [Liu et al., 2023b], with minor stylistic modifications for consistent representation. This work formalises the FF-A hypercall API for memory sharing between virtual machines in a virtualisation environment, and introduces VMSL, an Iris-based program logic to reason about virtual machines invoking the hypercalls. VMSL supports VM-local reasoning, akin to thread-local reasoning of CSL. On the top of the logic, the intended isolation property of the hypercall API is captured as *robust safety* via a logical relation. This result enables one to preserve the verification result of a VM when it runs together with other unknown and potentially malicious VMs.

Chapter 5 contains the content of [Liu et al., 2024], with minor corrections and modifications for consistent styling. This work includes instances of AxSL, a generic Iris-based CSL framework that can be instantiated with different memory models. Notably, one of its instances, AxSL^{Arm} is the first logic that supports thread-local reasoning for the very relaxed Arm-A memory model. AxSL^{Arm} supports reasoning about Arm's acyclic ordered-before (ob) relation, which is challenging as it suffers from a circularity issue. AxSL overcomes this challenge through an innovative semantic model and a novel two-phase adequacy proof, ensuring sound obreasoning. Furthermore, AxSL is built on *opax* semantics, a novel operationalisation of axiomatic models that provides a general approach for reasoning about axiomatic models in Iris. This work significantly extends the original AxSL paper [Hammond et al., 2024]; hence, this chapter also subsumes its content.

Chapter 6 contains the content of [Liu et al., 2025]. This work introduces AxSL+, a generalisation of AxSL, that supports *mixed-order* reasoning of multiple acyclic orders. Mixed-order reasoning is necessary for building high-level abstractions that improve verification tractability. AxSL+ implements it with a novel level-indexing semantic model, and an advanced adequacy proof. To demonstrate the benefits of mixed-order reasoning, the work presents AxSL+^{NA}, a simple Arm-A instance of AxSL+ that supports a high-level abstract assertion for non-racy locations, leveraging both ob and the coherence order eco of Arm-A

Part II

Publications

4

VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A

Abstract

Thin hypervisors make it possible to isolate key security components like keychains, fingerprint readers, and digital wallets from the easily-compromised operating system. To work together, virtual machines running on top of the hypervisor can make hypercalls to the hypervisor to share pages between each other in a controlled way. However, the design of such hypercall ABIs remains a delicate balancing task between conflicting needs for expressivity, performance, and security. In particular, it raises the question of what makes the specification of a hypervisor, and of its hypercall ABIs, good enough for the virtual machines. In this paper, we validate the expressivity and security of the design of the hypercall ABIs of Arm's FF-A. We formalise a substantial fragment of FF-A as a machine with a simplified ISA in which hypercalls are steps of the machine. We then develop VMSL, a novel separation logic, which we prove sound with respect to the machine execution model, and use it to reason modularly about virtual machines which communicate through the hypercall ABIs, demonstrating the hypercall ABIs' expressivity. Moreover, we use the logic to prove *robust safety* of communicating virtual machines, that is, the guarantee that even if some of the virtual machines are compromised and execute unknown code, they cannot break the safety properties of other virtual machines running known code. This demonstrates the intended security guarantees of the hypercall ABIs. All the results in the paper have been formalised in Coq using the Iris framework.

4.1 Introduction

A verification effort can only ever be as good as the specification it relies on. This is especially true for key security components like hypervisors, where a single error in design can void all security guarantees. Specifications for real-world programs are sizeable programs themselves, and thus commonly suffer from bugs themselves; and while some are found during the verification effort [Nienhuis et al., 2020, §VI], this is not always the case [Chidambaram, 2018]. Moreover, the verification effort does not necessarily validate the expressivity of the specification either. To address this, specifications themselves need to be validated and tested, in particular by exercising them to verify client code. In the terminology of DeepSpec, we need to make sure that specifications are 'live' [Appel et al., 2017], in that they are "connected via machine-checkable proofs to [not just] the implementation [but also to] client code".

In this paper, we formalise and validate a substantial fragment of the hypercall (aka 'hypervisor call', HVC) ABIs of FF-A, the Arm Firmware Framework for Arm A-profile [Arm Ltd., 2022], as implemented by Google's Hafnium hypervisor [Hafnium development team, 2022]. The hypercall ABIs allow virtual machines (VMs) running atop of a hypervisor to communicate and share data, e.g., by sending messages or by controlled sharing of memory pages, and to pass control to others. Our formalisation simplifies the ABIs compared to the informal FF-A specifications, but still captures the essence (see Section 4.2.1 for details). We then validate it by exercising it to verify key scenarios of VMs using the ABIs for controlled sharing of memory in the presence of adversarial, unknown code. Controlled sharing is essential for communication between VMs in real use cases, but makes the security analysis of hypervisors much more challenging.

Our running example is that of Figure 4.1, where the 'primary' VM (typically, Linux) is privileged, and can ask the hypervisor to schedule other, 'secondary' VMs (typically, the keychain, or DRMs). Here, we have two secondary VMs: one running known code, VM1, and one adversarial, running unknown code, VM2; each VM has its own pages, disjoint from those of the others. The primary VM, VM0, first asks the hypervisor to share one of its pages with VM1; then asks the hypervisor to run the adversarial VM2; and, when given back control, asks the hypervisor to run the known VM1.

Dealing with the HVC ABIs and their underlying use of virtual memory adds many components to the machine state: page tables, in-flight memory sharing transactions between VMs, etc. Managing the size and details of such a machine state poses a significant proof engineering challenge. For reasoning to be tractable, we need to be able to reason about known VMs individually: we should only need to consider the relevant parts of the machine state, and only need to take interference into account at interaction points, not at every step of the program.

To this end, we develop VMSL, a novel higher-order separation logic that supports formal modular reasoning about the execution of communicating VMs. VMSL effectively reduces the problem of verifying VMs communicating via the hypercall ABIs of FF-A to well-studied problems: cooperative multitasking, and functional

correctness of assembly.

One key intuitive desired security guarantee is *robust safety*: no matter what HVCs the adversarial VM2 may invoke, it will not be able to affect the private pages of VM0 and VM1, nor the page shared between only VM0 and VM1. This requires carefully designed ABIs, posing constraints to each HVC, making sure the desired guarantee is not breakable in any case, which results in a sophisticated and lengthy informal FF-A specification [Arm Ltd., 2022]. In this paper, we describe how to capture robust safety formally, even in the presence of in-flight transactions between VMs, and how to prove that the ABI specifications enforce robust safety.

We highlight the following features of our VMSL logic:

- VMSL is foundational [Appel, 2001]: we mechanise the definition of VMSL and prove it sound in Coq using the Iris separation logic framework [Jung et al., 2018b] and the Iris Proof Mode [Krebbers et al., 2017b]. Both the definition of VMSL and the examples using VMSL extensively rely on the expressive power of Iris to make reasoning about low-level code tractable, and we point out where we utilise it throughout the paper.
- VMSL supports modular reasoning in the sense that each VM can be verified individually. This is crucial for formal verification to work at scale.
- VMSL features two compatible logical resource sharing mechanisms to support reasoning about communication between VMs: (1) standard separation logic invariants, and (2) our *resumption condition*s, a logical sharing mechanism that offers more convenience than standard invariants for communication between VMs in the style of cooperative multitasking.
- VMSL is factored in two parts: a general part that handles issues that arise for any low-level model with scheduling, and a specific part that deals with the HVC ABIs of FF-A.
- VMSL is sufficiently expressive to support not only formal reasoning about concrete known programs, but also the definition of so-called logical relations which can be used to reason about robust safety. We use logical relations to reason about scenarios like that of Figure 4.1, where some VMs run known code and others run unknown possibly adversarial code.

Contributions

- We formalise a substantial fragment of the Arm's FF-A ABIs, as implemented by Hafnium, in the form of an operational semantics in which HVCs are primitive steps (Section 4.2).
- We develop and prove soundness of VMSL, a novel separation logic for modular reasoning about communicating VMs (Section 4.3).
- We show how we capture the desired security guarantees using logical relations, and how we apply them to reason about robust safety (Section 4.4).

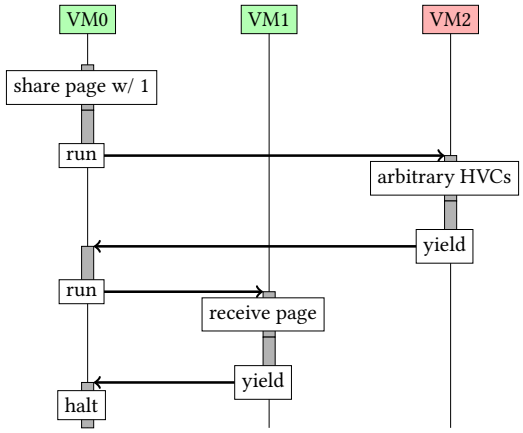


Figure 4.1: A motivating example where a compromised VM2 is contained: the page sharing between VM0 and VM1 is guaranteed to succeed if the adversarial VM2 yields, no matter what other HVCs VM2 makes. The memory integrity of the page is guaranteed.

All of our results are mechanised in Coq using Iris. The Coq formalisation and the instructions of usage are available in the supplementary material [Liu et al., 2023a].

Non-goals We focus on exercising the HVC ABIs, and thus do not address other key complementary aspects, which we discuss in Section 4.5. In particular:

1. We are not verifying a hypervisor, but rather making sure that the hypervisor specification that we are providing is adequate.

2. We focus on the HVC ABIs, and our operational semantics is a minimalistic instruction set: it has the right shape, but it is far from a full-scale ISA.

3. Our operational semantics does not include interrupts, and assumes that there is no concurrency, as characterising the semantics of virtual memory in a concurrent setting is work in progress [Simner et al., 2022].

Threat model We only consider integrity, not secrecy. Our attacker model is that of adversary VMs running unknown code; we do not consider side-channels. To reason about adversarial VMs running unknown code, we only assume knowledge of initially accessible pages and transactions related to adversaries; both the content of memory and registers of adversaries are unspecified. Adversaries therefore could perform attacks by executing malicious code stored in their memory. For instance, adversaries could invoke arbitrary HVCs to try to interfere with in-flight transactions between trusted VMs, or read/write memory of other VMs. With this model, we show that adversaries cannot break the integrity of memory under protection of hardware and the hypervisor.

4.2 Formalising A Substantial Fragment of the HVC ABIs

As we focus on the HVC ABIs, we use a simplified subset of the Arm-A instruction set, with only one unusual feature: the hvc instruction. Figure 4.2 shows the running example of Figure 4.1 more precisely in our language.

4.2.1 Scope

We specify the hardware behaviours of virtualisation, including page table lookup and context switching, plus the following HVCs of FF-A:

1. for memory sharing: Donate, Lend, Share, Retrieve, Relinquish, and Reclaim;
2. for scheduling: Run, Yield, and Wait; and
3. for messaging: (asynchronous) Send and Poll.

This covers most of FF-A, apart of the 'secure world' trusted computing functionality involving TrustZone. We omit the synchronous variant of Send, which requires extra machinery without increasing expressivity, and the new messaging HVC, notify, that was introduced after this work started.

Simplification We make two main simplifications in our model of FF-A:

1. We only formalise the ownership and access fields of the page table entries, and only consider read-write permissions.
2. We only model 1-to-1 sharing (as implemented by Hafnium) instead of 1-to-n, and accordingly simplify the format of transaction descriptors.

These, along with other minor simplifications, help keep the size of our model manageable, but do not significantly omit specification details or impact expressivity. For instance, we believe the model can be adapted to support 1-to-n sharing.

Conformance As with any formal modelling activity, there is an unavoidable gap between the informal FF-A specification and our formal specification. We have tried to follow the intent of the informal specification when designing our formal model, and cross-referenced it with the Hafnium implementation of the informal spec to gain more confidence in our formal model. Future work includes showing that some of the Hafnium HVC implementations *refine* our formal model.

4.2.2 Formalising HVCs

Informally, a hypervisor provides the illusion to VMs that they are running on a machine in which the whole HVC is just a step of the machine; the hypervisor itself is invisible. Accordingly, in our model, an HVC is a primitive step of the operational semantics. The reduction rule for a Share in Figure 4.3 is a representative example, and we explain it below.

```
1  /* VM0 */
2  /* save x to p */
3  mov R5 <- #p
4  str R0 [R5]
5  /* prepare desc */
6  mov R5 <- #ptx
7  mov R4 <- 0
8  str R4 [R5]
9  ...
10 /* share p */
11 mov R0 <- #Share
12 mov R1 <- 4
13 hvc
14 /* send handle */
15 mov R5 #ptx
16 str R2 [R5]
17 mov R3 <- R2
18 mov R0 <- #Send
19 mov R1 <- 1
20 mov R2 <- 1
21 hvc
22 /* run VM2 */
23 mov R0 <- #Run
24 mov R1 <- 2
25 hvc
26 /* run VM1 */
27 mov R0 <- #Run
28 mov R1 <- 1
29 hvc
30 /* read x */
31 mov R1 <- #p
32 ldr R0 [R1]
33 halt
```

```
1  /* VM1 */
2  /* fetch handle */
3  mov R5 <- #rx
4  ldr R4 [R5]
5  mov R0 <- #MsgPoll
6  hvc
7  /* retrieve p */
8  mov R1 <- R4
9  mov R0 <- #Retrieve
10 hvc
11 /* x = x+2 */
12 mov R5 <- #p
13 ldr R3 [R5]
14 add R3 2
15 str R5 [R3]
16 /* yield */
17 mov R0 <- #Yield
18 hvc
```

Figure 4.2: Code of the two known VMs in Figure 4.1. Additional notation is added to improve readablity. Symbols with prefix # are constant values: x is the data stored in R0 that VM0 will share with VM1; p is the page that VM0 will share (represented with the base address of the page); *ptx* is the base addresses of the write-only messaging buffer (TX) of VM0, and *prx* is the read-only buffer (RX) of VM1. We assume that the two programs live at the start of two separate pages, pp_0 and pp_1.

Memory Access

On a concrete machine, an hvc causes a jump to a higher exception level and the execution of hypervisor code. The hypervisor code operates on its private data in physical memory; in our model, the private state of the hypervisor is represented abstractly, separate from the physical memory that the VMs operate on, which we model as a partial function from memory addresses to machine words (both are represented by our type of machine words, *Word*).

In particular, on a concrete machine, the page tables are in-memory data structures that are edited by the hypervisor and looked up by the hardware; in our model, the page tables are merged into one partial (mathematical) function that is updated by memory-sharing HVCs. The partial function maps a page identifier (page base address, which is sufficient, given that we assume identity address mappings) to a *page status*, which is composed of an optional page owner, the set of *VMID*s of the VMs that have access to the page, and a bit indicating whether it is exclusively owned (can only be accessed) by one VM. For instance, the status of page p in the example of Figure 4.2 is initially $(\mathsf{Some}(0), \{0\}, \mathsf{True})$, since VM0 has exclusive ownership on the page; and it is updated to $(\mathsf{Some}(0), \{0, 1\}, \mathsf{False})$ after the page is shared with VM1.

When a VM with VMID i tries to perform a memory access at an address a, e.g. str at line 4 storing the value in R5 to address p, the page status of the page p is

$$\frac{\begin{array}{c}\sigma.\mathtt{curr} = i \\ \mathit{valid\_instr}(\sigma, i) = \mathtt{Some}(\mathsf{hvc}, a) \qquad \mathit{valid\_share}(\sigma, i) = \mathtt{Some}(i_r, s, h) \\ \sigma' = \left\{ \begin{array}{l} \mathtt{mem} = \sigma.\mathtt{mem}; \quad \mathtt{curr} = \sigma.\mathtt{curr}; \quad \mathtt{mb} = \sigma.\mathtt{mb}; \\ \mathtt{pgt} = \sigma.\mathtt{pgt}\left[\begin{array}{l} p \mapsto (\mathtt{Some}(i), \{i\}, \mathsf{False}) \\ \mid (p \in s) \end{array} \right]; \\ \mathtt{regs} = \sigma.\mathtt{regs}[i]\left[\begin{array}{l} \mathsf{pc} \mapsto a + 1; \\ \mathsf{R0} \mapsto \mathit{encode}(\mathsf{Succ}); \\ \mathsf{R2} \mapsto h \end{array} \right]; \\ \mathtt{trans} = \sigma.\mathtt{trans} \\ \qquad [h \mapsto \mathtt{Some}((i, i_r, s, \mathsf{Share}), \mathsf{False})]; \end{array} \right\} \end{array}}{(\mathtt{Normal}, \sigma) \rightarrow (\mathtt{Normal}, \sigma')}$$

Figure 4.3: Reduction rule for Share

looked up in the page table, and checked to determine whether the VM is allowed to access p (which it can in this case, since 0 is an element of the 'accessible' set $\{0\}$). If the access is not allowed by the page table, a page fault is raised. In our setup, this terminates the execution (of all VMs, because there is no concurrency) with the execution mode `PageFault`, and therefore a page fault is *safe*.

Configuration

A *configuration* is a pair of a *state* together with an *execution mode*. A *state* of our operational semantics is composed of the aforementioned components for modelling memory access, plus those for HVCs:

$$\mathit{State} \stackrel{\text{def}}{=} \left\{ \begin{array}{llll} \mathit{mem} & : \mathit{Word} \rightharpoonup \mathit{Word}; & \mathit{pgt} & : \mathit{PageID} \rightharpoonup \mathit{PageStatus}; \\ \mathit{regs} & : \mathit{VMID} \rightarrow \mathit{RegisterFile}; & \mathit{curr} & : \mathit{VMID}; \\ \mathit{trans} & : \mathit{Transactions}; & \mathit{mb} & : \mathit{VMID} \rightarrow \mathit{Mailbox}; \end{array} \right\}$$

We have three execution modes: `Normal`, `PageFault`, and `Halted`.

The machine can only take a further step to execute the next instruction if it is in `Normal` mode. `Halted` is the mode reached by 'normal' termination via the halt instruction, and, as stated above, `PageFault` is used for page faults.

Transactions

On a concrete machine, to support memory sharing transactions between VMs, the hypervisor needs to maintain some metadata in its private memory; in our model, we keep a partial mapping from transaction handles (machine words) to abstract *transactions*, which are composed of the sender, the receiver, the set of pages being sent, the type of the transaction, and the state of it (a bit indicating whether the receiver has retrieved the access to the pages). For instance, the hvc at

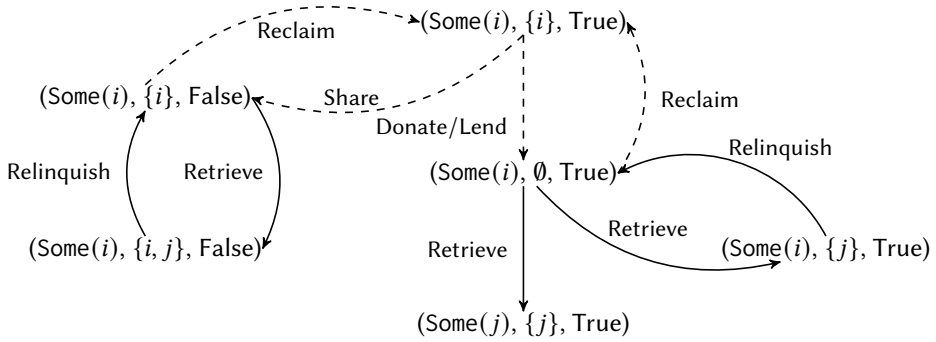


Figure 4.4: The state transition system of the status of a page during a transaction. HVCs with dashed arrows are allowed for the sender i, and others are allowed for the receiver j.

line 13 of VM0 invokes a sharing transaction of page p to VM1, which is represented as $\mathsf{Some}((0, 1, \{p\}, \mathsf{Share}), \mathsf{False})$ (see the last line of antecedents in the rule in Figure 4.3).

A VM is allowed to send pages to other VMs via transactions. To do so, the sending VM first has to prepare a transaction descriptor specifying the receiver and the page IDs of the pages in its TX page (lines 5–9 in the example). Next, the sending VM invokes a memory sending HVC, asking the hypervisor to create a transaction of the type given by the descriptor. The type of transaction (donation, sharing, or lending) determines the effect of the HVC on the status of the pages being sent, as per Figure 4.4. The sharing of page p in the example corresponds to edges ❷ and ❺. In all cases, the hypervisor checks that the pages are owned and exclusively accessible by the sender before creating the transaction (e.g. done by *valid_share* in Figure 4.3). If the checking fails, the hypervisor returns an error code to the VM and resumes its execution. If it succeeds, the hypervisor then returns a fresh handle h (initially mapped to None, meaning that it is not bound to any transaction) referring to the newly created transaction to the sender, and remembers the transaction in its metadata (*trans*).

VMs can invoke other HVCs with the same handle to refer to the transaction. For instance, with the hvc at line 10, VM1 Retrieves access to the page, flipping the retrieved bit to True. In case of donation, this HVC also transfers ownership of the pages to the receiver and finishes the transaction (and frees the handle). In case of sharing or lending, the receiver could Relinquish access to the pages afterwards, flipping the bit back. The sender can Reclaim exclusive access to the pages if the access has not been retrieved, or has been relinquished by the receiver (in either case, the retrieved bit is False), which is the second way of ending the transaction.

Scheduling

On a concrete machine, to support switching between VMs, the hypervisor needs to save registers by spilling them in its private memory, and restore them upon context switching; in our model, we keep a total mapping (*regs*) from VMIDs to *register files*, where a register file is itself a map from register names to words, and a VMID (*curr*) to remember which VM is currently running.

By duplicating *RegisterFile* and picking the right one to update according to *curr* when registers are modified, we avoid modelling register saving and restoring at context switching. For instance, mov at line 3 of VM0 only updates R5 of VM0 since *curr* is 0. As a consequence, the switching HVCs only need to change *curr*.

FF-A allows putting the responsibility of scheduling VMs either on the hypervisor, or delegates it to VM0, the 'primary' VM. Typically, thin hypervisors like Hafnium choose the latter, for instance letting the thread scheduler of Linux make scheduling decisions. We model the latter use case.

Therefore, it grants the primary VM the privilege to Run other so-called secondary VMs. Secondary VMs are only allowed to return control back to the primary; either explicitly with Yield, or as the consequence of an HVC, for example to wait for a message with Wait.

Messaging

To support messaging between VMs, on a concrete machine, the hypervisor needs to maintain two dedicated memory pages, named TX and RX, as the message buffers for each VM, and remembering the state of all RX buffers (e.g. whether the buffer is full); in our model, we keep a total mapping (*mb*) from *VMID* to *Mailbox*, which consists of two buffers.

The TX and RX buffers are respectively write-only and read-only, and are used for sending and receiving messages between two VMs, or a VM and the hypervisor. Line 21 of VM0 Sends the handle referring to the sharing transaction to VM1. The hypervisor copies the handle from the TX page of VM0, pastes it to the RX page of VM1, and remembers the length and the sender in its private state as `Some(1, 0)`. In the case where the sender is a secondary VM, the control is yielded to the primary immediately, notifying it that a message has just been sent to the receiver, so that the primary can schedule the receiver to run next to actually receive the message. The receiver, like VM1, can ask for the length and the sender of the message with Poll (line 6 of VM1), which also notifies the hypervisor that it is ready to take the next message (updates the RX buffer to `None`).

Calling Convention

The calling convention that we have used in the example above works in general as follows: to invoke a specific HVC, a VM executes the hvc instruction with the identifier of the HVC in R0, and other arguments saved in successive general-purpose registers (for example, the identifier of the VM to Run in R1), or in the TX buffer

(for "large" arguments like transaction descriptors), as appropriate. Return values, including whether the HVC is successful and possible error codes, are passed back to the VM via return registers, like in Figure 4.3, or RX buffers (depending on the HVC).

4.3 Reasoning about Communicating VMs

To validate our model of the FF-A HVC ABIs, we develop VMSL, a program logic designed to reason about key scenarios of VMs communicating using the FF-A ABIs. We start this section by discussing two of the key challenges involved in developing a program logic for communicating VMs.

The programs running on VMs are imperative and operate on mutable shared data and so we base VMSL on separation logic [Reynolds, 2002]. In particular, this will allow us to support *local reasoning* via the *frame rule* of separation logic, as we show below.

The first challenge is that we wish to reason about a low-level language model where instructions are stored in the memory, which complicates the formulation of a sequential composition proof rule, which usually makes it possible to reason about instructions one at a time. This is a common problem, and we neatly capture a 'folklore' solution in a small Iris library in the form of *single-step weakest preconditions*. We discuss how our approach relates to previous work on program logics for assembly in Section 4.5.

The second key challenge is that we wish to support 'VM-local' reasoning: it should be possible to verify each VM individually. This is analogous to 'thread-local' reasoning in concurrent separation logic, and is crucial for formal verification to work at scale. We could treat each VM in a manner similar to how a thread is treated in concurrent separation logic, and then use concurrent separation logic style *invariants* to reason about sharing of data among different VMs. However, such invariants were designed for concurrency, and pose an undue burden in our setting where VMs are executed sequentially but not concurrently. Therefore, we introduce *resumption conditions*, an alternative mechanism to share resources among VMs, which allows a VM to use shared resources freely during its execution until control is transferred to another VM.

We explain our solutions to the two challenges on our example in Section 4.3.2; motivate and describe them in more detail in Section 4.3.3. With the solutions implemented in VMSL using Iris, we prove soundness of the logic with respect to the operational semantics of the machine model. All of VMSL's proof rules are sound with respect to our definition of weakest preconditions, and we have proven an *adequacy theorem* which intuitively says that if a weakest precondition holds in the VMSL, then it really means that it is safe to execute the program on the machine. We refer the reader to our Coq formalisation for a precise formal statement of the soundness and adequacy theorems and the proofs thereof.

$$\text{SS-MOV}\quad \frac{❶\mathsf{pc}@i \overset{\text{reg}}{\longmapsto} a * ❷a \in_p s * ❸\mathsf{Pgt}@i \overset{\text{acc}}{\longmapsto} s * ❹a \overset{\text{mem}}{\longmapsto} \mathit{encode}(\mathsf{mov}\ r\ n) * ❺r@i \overset{\text{reg}}{\longmapsto} -}{\mathsf{sswp}\ \mathtt{Normal}\ @\ i \left\{ \begin{array}{l} (\mathsf{False}, \mathtt{Normal}). \\ \left(\begin{array}{l} \mathsf{pc}@i \overset{\text{reg}}{\longmapsto} a+1 * a \overset{\text{mem}}{\longmapsto} \mathit{encode}(\mathsf{mov}\ r\ n) * \\ \mathsf{Pgt}@i \overset{\text{acc}}{\longmapsto} s * r@i \overset{\text{reg}}{\longmapsto} n \end{array} \right) \end{array} \right\}}$$

Figure 4.5: The proof rule for an immediate-to-register mov instruction. The updated resources are highlighted in yellow as in later rules. For simplicity, we omit the *encode* function that maps non-words including instructions and HVC identifiers to *Word*s in later rules. Also, we use $\mathsf{IsInstr}@i(s, a, \mathsf{mov}\ r\ n)$ to represent that the mov instruction is stored at address a which belongs to the page that is one of VMi's accessible pages s.

4.3.1 VMSL

In this section we introduce VMSL by explaining how it is used to specify and reason about VMs executing *known* code. We use a simplified variant of Figure 4.2 without invoking the unknown VM2 (that is, with lines 22–25 of VM0 removed) as a running example.

Informal Specification

In this example, the primary VM writes the content x of register R0 to the first location of page p, shares the page with VM1, then schedules VM1. VM1 retrieves access to the page p, increments the first location of p by two, then yields. The primary VM then reads from p into R0, and halts. We want to show that it reads $x + 2$.

Points-to Assertions

To state this formally, we introduce the classic register 'points-to' assertion, $r@i \overset{\text{reg}}{\longmapsto} v$, which captures the fact that register r contains the value v; because our registers are banked, we specify which VM the register belongs to via its VMID, i. As usual in separation logic, our assertion also captures ownership of register r of VMID i, so that this assertion is exclusive. In Table 4.1, we present a collection of similar points-to predicates of VMSL, together with their intuitive meanings. We introduce most of them gradually along with our explanation of how we use VMSL to reason about the example.

Table 4.1: Selected collection of resources of VMSL

Predicate	Intuition
$r@i \overset{\text{reg}}{\longmapsto} w$	register r of VMi contains word w
$a \overset{\text{mem}}{\longmapsto} w$	value w is at location a
$\mathsf{Pgt}@i \overset{\text{acc}}{\longmapsto} s$	VMi has access to pages s
$\mathsf{Pgt}@p \overset{\text{own}}{\longmapsto} i$	VMi owns page p
$\mathsf{Pgt}@p \overset{\text{excl}}{\longmapsto} i$	VMi's access to page p is exclusive
$\mathsf{Tran}@h \overset{\text{tran}}{\longmapsto} t$	transaction t is bound to handle h
$\mathsf{Tran}@h \overset{\text{rtrv}}{\longmapsto} b$	status of transaction bound to h is b
$\mathsf{Mb}@i \overset{\text{rx}}{\longmapsto} p$	VMi's RX page is p
$\mathsf{Mb}@i \overset{\text{tx}}{\longmapsto} p$	VMi's TX page is p
$\mathsf{MemPage}(p, ws)$	content of page p is ws
$\mathsf{FreshHandles}(hs)$	handles hs are fresh

Formal Specification

Returning to the example, starting from a state where $\mathsf{R0}@0 \overset{\text{reg}}{\longmapsto} x$, with other resources and some side conditions we introduce below, we want to show that, when the machine terminates, VM0 reaches a `Halted` state (indicating success), and moreover we have $\mathsf{R0}@0 \overset{\text{reg}}{\longmapsto} x + 2$. We phrase this in VMSL by using a *weakest precondition* predicate $\mathsf{wp}\ m\ @\ i\ \{Q\}$ which expresses the partial correctness of the VMi, i.e., we execute the VM with mode m and, if it terminates, then the postcondition Q holds:

$$\mathsf{R0}@0 \overset{\text{reg}}{\longmapsto} x * \ldots (\text{other resources})$$
$$\vdash \mathsf{wp}\ \texttt{Normal}\ @\ 0\ \left\{\texttt{m.m} = \texttt{Halted} * \mathsf{R0}@0 \overset{\text{reg}}{\longmapsto} x + 2\right\}$$

4.3.2 Proving the Specification

First Instruction

To safely execute the first instruction of VM0, mov R5 #p (where p is an immediate), we need, as captured in our SS-MOV proof rule for an immediate-to-register mov, to know/show:

1. The value a of the program counter, which indicates the location of the current instruction in the memory, as captured by the points-to for registers $\mathsf{pc}@i \overset{\text{reg}}{\longmapsto} a$ (here, $\mathsf{pc}@0 \overset{\text{reg}}{\longmapsto} pp_0$).
2. Knowledge that the page at address a (here, pp_0) is in the accessible set s of *PageID*s...

SS-SHARE

$$\frac{\begin{array}{l}\textcircled{1}\mathit{ValidDesc}(\mathit{memtx}, i, j, \mathit{ps}) \land \textcircled{2}\mathit{ps} \subseteq s \land \textcircled{3}\mathit{hs} \neq \emptyset \land \mathsf{IsHVC@}i(s, a, \mathsf{Share}) * \\ \mathsf{R1@}i \overset{\text{reg}}{\longmapsto} l * \mathsf{R2@}i \overset{\text{reg}}{\longmapsto} - * \textcircled{4}\mathsf{Mb@}i \overset{\text{tx}}{\longmapsto} \mathit{ptx} * \textcircled{5}\mathsf{MemPage}(\mathit{ptx}, \mathit{memtx}) * \\ \textcircled{6} \mathop{\Large *}_{p \in \mathit{ps}} (\mathsf{Pgt@}p \overset{\text{own}}{\longmapsto} i * \mathsf{Pgt@}p \overset{\text{excl}}{\longmapsto} \mathsf{True}) * \textcircled{7}\mathsf{FreshHandles}(\mathit{hs})\end{array}}{\mathtt{sswp}\ \mathtt{Normal}\ @\ i \left\{ \begin{array}{l}(\mathsf{False}, \mathtt{Normal}). \\ \left(\begin{array}{l}\mathsf{pc@}i \overset{\text{reg}}{\longmapsto} a + 1 * a \overset{\text{mem}}{\longmapsto} \mathsf{hvc} * \mathsf{Pgt@}i \overset{\text{acc}}{\longmapsto} s * \\ \mathsf{R0@}i \overset{\text{reg}}{\longmapsto} \mathsf{Succ} * \mathsf{R1@}i \overset{\text{reg}}{\longmapsto} l * \\ \mathsf{Mb@}i \overset{\text{tx}}{\longmapsto} \mathit{ptx} * \mathsf{MemPage}(\mathit{ptx}, \mathit{memtx}) * \\ \mathop{\Large *}_{p \in \mathit{ps}} \mathsf{Pgt@}p \overset{\text{own}}{\longmapsto} i * \mathsf{Pgt@}p \overset{\text{excl}}{\longmapsto} \mathsf{False} * \\ \exists h.\, h \in \mathit{hs} \land \mathsf{R2@}i \overset{\text{reg}}{\longmapsto} h * \mathsf{FreshHandles}(\mathit{hs} \setminus \{h\}) * \\ \mathsf{Tran@}h \overset{\text{tran}}{\longmapsto} (i, j, \mathit{ps}, \mathsf{Share}) * \mathsf{Tran@}h \overset{\text{rtrv}}{\longmapsto} \mathsf{False}\end{array} \right)\end{array} \right\}}$$

SS-RUN

$$\frac{\begin{array}{l}\textcircled{1} i \neq 0 \land \mathsf{IsHVC@}0(s, a, \mathsf{Run}) * \mathsf{R1@}0 \overset{\text{reg}}{\longmapsto} i * \textcircled{2}\mathsf{RC}_{1/2}\mathsf{@}i\,\{\Psi_i\} * \textcircled{3}\mathsf{RC}_1\mathsf{@}0\,\{-\} * \\ \textcircled{4}\left(\left(\begin{array}{l}\mathsf{pc@}0 \overset{\text{reg}}{\longmapsto} a + 1 * a \overset{\text{mem}}{\longmapsto} \mathsf{hvc} * \mathsf{Pgt@}0 \overset{\text{acc}}{\longmapsto} s * \\ \mathsf{R0@}0 \overset{\text{reg}}{\longmapsto} \mathsf{Run} * \mathsf{R1@}0 \overset{\text{reg}}{\longmapsto} i * \Phi_{\mathit{othr}} * \mathsf{RC}_1\mathsf{@}0\,\{\Psi_0\}\end{array} \right) \mathrel{-\!\!*} \Psi_i * \Phi_{\mathit{rest}} \right) * \textcircled{5}\Phi_{\mathit{othr}}\end{array}}{\mathtt{sswp}\ \mathtt{Normal}\ @\ 0\ \left\{(\mathsf{True}, \mathtt{Normal}).\, \mathsf{RC}_{1/2}\mathsf{@}0\,\{\Psi_0\} * \Phi_{\mathit{rest}}\right\}}$$

WP-SSWP

$$\mathtt{wp}\ \mathtt{m}\ @\ i\ \{\Phi\} \dashv\vdash \mathtt{sswp}\ \mathtt{m}\ @\ i \left\{ (b, \mathtt{m}'). \begin{array}{c}((b \land \mathsf{RCHolds@}i) \lor (\neg b)) \mathrel{-\!\!*} \\ \mathtt{wp}\ \mathtt{m}'\ @\ i\ \{\Phi\}\end{array} \right\}$$

RC-HOLD

$$\mathsf{RCHolds@}i * \mathsf{RC}_{1/2}\mathsf{@}i\,\{\Psi\} \vdash \triangleright \Psi * \mathsf{RC}_1\mathsf{@}i\,\{\Psi\}$$

Figure 4.6: Selected rules of VMSL

❸ ...that are mapped for the current VM, as captured by ownership of the page tables points-to assertion, $\mathsf{Pgt@}i \overset{\mathrm{acc}}{\longmapsto} s$ (here, $\mathsf{Pgt@0} \overset{\mathrm{acc}}{\longmapsto} s$).

❹ Ownership of the memory points-to resource for that memory location, $a \overset{\mathrm{mem}}{\longmapsto} w$ (here, $pp_0 \overset{\mathrm{mem}}{\longmapsto} w$), which contains a word w that is the encoding of an immediate-to-register mov instruction (here, mov R0 #p).

❺ Ownership of the register points-to resource for the affected register (here, $\mathsf{R5@0} \overset{\mathrm{reg}}{\longmapsto} -$); we do not need to know what it contains (as signified by the use of $-$), but we must have the right to update it.

After the mov instruction, the VM does not lose control (so the switching bit is False), and the execution mode is still `Normal`. We get the updated resources back in our context; in particular, the program counter has been incremented, $\mathsf{pc@0} \overset{\mathrm{reg}}{\longmapsto} pp_0 + 1$, and the register now contains the immediate, $\mathsf{R5@0} \overset{\mathrm{reg}}{\longmapsto} p$; the page tables and the instruction have not been affected, so we get their assertions back unchanged.

The proof rule requires exactly the resources needed to safely execute the instruction; other resources are implicitly kept unchanged via framing, which is a key feature of separation logic that saves us from maintaining global resources all the time, and helps keep the proof effort manageable.

The SS-MOV rule, and all other single-instruction proof rules, use *SSWP*, our single-step variant of weakest preconditions. A *single-step weakest precondition* captures an intuitive idea (see Section 4.5): it is like a weakest precondition that only specifies the behaviour of a single step (an instruction). Applying a single-step weakest precondition takes resources specified in the premise, and returns resources stated in the postcondition, with the resulting execution mode and a bit indicating whether the instruction would cause the VM to lose control of the machine (the hypervisor switching to another VM to execute). Single-step weakest preconditions allow us to reason about one instruction at a time. We show how to formally apply it to *weakest precondition* in Section 4.3.3.

Sharing

The following instructions prepare the descriptor and arguments for the Share HVC at line 13. They only involve register manipulations, which can be reasoned about in a similar way to the first instruction, and memory accesses. To reason about memory access instructions, including ldr and str, we need memory points-to predicates, with side conditions checking whether the VM has the permission to access the address, similar to ❷ of SS-MOV.

Before reasoning about this specific Share, let us first consider the expected behaviour of a general Share HVC, specified by the SS-SHARE rule. To share pages represented by a set of PageIDs *ps*, VM*i* invokes a Share HVC with a descriptor in its TX page describing information about the transaction. Therefore, the proof rule requires ❹ the TX page *ptx*; ❺ ownership of the page with content *memtx*, which is expressed as memory points-tos for all locations of the page, connected by $*$; and ❶

knowledge that the descriptor stored in *memtx* is valid. In addition, after validating the descriptor, the page table is examined to check whether VMi is allowed to share those pages in *ps*. Therefore, the rule requires ❻ page ownership $\mathsf{Pgt}@p \overset{\text{own}}{\longmapsto} i$ and exclusiveness $\mathsf{Pgt}@p \overset{\text{excl}}{\longmapsto} \mathsf{True}$ to VMi of each page p in *ps*. The side condition ❷ plus the resource for page access (included in IsHVC) further ensure that VMi has access to those pages. This information, combined, ensures that VMi is allowed to share pages *ps*. To initiate a transaction, the hypervisor has to allocate a fresh transaction handle h, which is ensured by ❼ remembering the set hs of available handles, and ❸ requiring hs not be empty. The hypervisor further binds h to the meta-information and the state of the transaction that are also represented as resources, as in the postcondition. It is worth-noting that in practice a predicate can be built upon these resources, e.g. TranHandles shown in Section 4.4, leveraging the resource separation to guarantee that fresh and allocated handles are disjoint, which would reduce the handle availability reasoning to easy-to-discharge set disjointness side goals.

In our example, VM0 shares a single page p to VM1, so we let i, j, and ps be 0, 1, and $\{p\}$ respectively. ❶ is justified by the previous instructions constructing the descriptor correctly. ❷ is justified as we assumed s to be $\{pp_0; p; p_{tx}\}$. ❸ is justified by assuming a non-empty hs in the specification. After applying the proof rule, we get $\mathsf{Tran}@h \overset{\text{tran}}{\longmapsto} (0, 1, p, \mathsf{Share})$ and $\mathsf{Tran}@h \overset{\text{rtrv}}{\longmapsto} \mathsf{False}$, stating that the requested transaction has been initiated, and is bound to h, which is also returned to VM0 so that it can refer to the transaction.

Messaging

To retrieve access to the shared page p, VM1 has to refer to the transaction with the handle h. To let VM1 do so, VM0 passes h to it by messaging at lines 14–21. Messaging essentially copies from the sender's TX page and pastes into the receiver's RX page; therefore, the proof rule for messaging requires the resources for the two pages and associated memory. We capture the state of VM1's RX page with a resource $\mathsf{RXState}@1 \mapsto \mathtt{Some}(1, 0)$ in the example, expressing that VM0 has passed one word to VM1.

Scheduling

At line 29, VM0 runs VM1 to allow VM1 to receive the handle and retrieve page p. To reason about such scheduling, we introduce a *resumption condition* for VM1. A *resumption condition* for a VMi, denoted as $\mathsf{RC}_1@i\,\{\Psi\}$, captures the resources Ψ that need to be handed over to VMi to resume its execution. We use *resumption conditions* to express communication protocols (reminiscent of session types [Honda et al., 2011; Yoshida and Gheri, 2020]) between VMs, and to transfer resources between VMs along the scheduling control flow. Accordingly, the proof rule for Run, SS-RUN, uses a *resumption condition*. Concretely, we have to show the following to apply SS-RUN when the primary VM, VM0, is about to run VMi:

1. The VM being run is not the primary VM itself.
2. VM0 has to satisfy the *resumption condition* of VMi, Ψ_i. The fraction $1/2$ indicates that the *resumption condition* is split into two halves, and only one half is required. We elaborate on this point later.
3. We may pick the *resumption condition* of VM0, Ψ_0, that VMi will have to satisfy to yield back.
4. The magic wand $P \mathrel{-\!\!*} Q$ is separation logic's resource-aware implication. It is used here to express that with resources required by the rule (the first line) and 5, we can show Ψ_i, intuitively the resources transferred to VMi, and the left over Φ_{rest}, i.e. the resources that are required by the rule, but not needed to show Ψ_i, that are still owned by VM0 afterwards.
5. Other resources required to justify Ψ_i.

By picking the right Ψ_i and Ψ_0, we describe the protocol according to which shared resources are transferred between the VMs. In our example, we know that to run VM1, VM0 has to have written x to the page p, shared the page, sent the handle, and run VM1. We express this in Ψ_i as follows:

$$\begin{aligned}\Psi_i \triangleq\ & p \overset{\text{mem}}{\longmapsto} x * \mathsf{Tran}@h \overset{\text{tran}}{\longmapsto} (0, 1, \{p\}, \mathsf{Share}) * \mathsf{Tran}@h \overset{\text{rtrv}}{\longmapsto} \mathsf{False} * \\ & \mathsf{Mb}@1 \overset{\text{rx}}{\longmapsto} prx * \mathsf{RXState}@1 \mapsto \mathsf{Some}(1, 0) * prx \overset{\text{mem}}{\longmapsto} h * \mathsf{R0}@0 \overset{\text{reg}}{\longmapsto} \mathsf{Run} * \\ & \mathsf{R1}@0 \overset{\text{reg}}{\longmapsto} 1 * \mathsf{RC}_{1/2}@0\,\{\Psi_0\}\end{aligned}$$

Note that when VM1 yields back control to VM0, it needs to have established VM0's resumption condition, so we also include $\mathsf{RC}_{1/2}@0\,\{\Psi_0\}$ in Ψ_i. VM1 thus can refer to Ψ_0 and show it when yielding. In our example, we want to show that VM1 has incremented x by 2 and yielded. We express this in Ψ_0:

$$\Psi_0 \triangleq p \overset{\text{mem}}{\longmapsto} x + 2 * \mathsf{R0}@0 \overset{\text{reg}}{\longmapsto} \mathsf{Yield} * \mathsf{R1}@0 \overset{\text{reg}}{\longmapsto} 1$$

To justify 4, we let Φ_{othr} be Ψ_i except for its last three assertions, and Φ_{rest} naturally be the resources that are in the premise but not required by Ψ_i.

We get Φ_{rest} and $\mathsf{RC}_{1/2}@0\,\{\Psi_0\}$ after applying the rule. To explain how to get resources stated in Ψ_0 out, we first introduce RCHolds@i. It assumes the resumption of VMi and can interact with the *resumption condition* of VMi by RC-HOLD. Intuitively speaking, the rule says that if we know the resumption condition of a VM, and the VM is indeed resumed, then the condition holds. $\triangleright\,\Psi$ means that Ψ holds *later*, i.e. after taking a step in the underlying model (this is used to break circularity of definitions [Jung et al., 2016, 2015]). Back to the example, we already get $\mathsf{RC}_{1/2}@0\,\{\Psi_0\}$ in the postcondition, so we would be able to apply this rule and proceed with the proof with the transferred-back resources in Ψ_0 if we have RCHolds@0 as well. For now, readers only need to know that we can actually get it for free, because we have baked it into the definition of weakest preconditions in a way that we can get it out when a switching just happened.

Halting and Suspension

After loading the word $x + 2$ at p to R0, the execution of VM0 is terminated by a halt. The proof rule updates the execution mode from `Normal` to `Halted`, and thus we obtain the postcondition of our initial specification, $\mathtt{m} = \mathtt{Halted} \land \mathrm{R0@0} \overset{\mathrm{reg}}{\longmapsto} x + 2$, and conclude the proof.

The proof of VM0 does not consider the code of VM1, due to the 'VM-modularity' of VMSL. All we needed was an abstract characterisation of the protocol governing the interaction between VM0 and VM1, as captured by the resumption conditions.

The proof of VM1 is similarly done without considering the code of VM0, but concludes in a different way, as VM1 does not terminate, but instead suspends via the Yield at line 18. Because our protocol specifies it will not be scheduled again, it suffices to show that when we resume it, we get an immediate contradiction.

4.3.3 More on Single-step Weakest Preconditions and Resumption Conditions

The example above shows how single-step weakest preconditions and resumption conditions are the two key components that make reasoning with VMSL manageable. We now discuss them in more detail, and point out how an expressive higher-order separation logic like Iris makes reasoning sound and tractable.

Single-step Weakest Preconditions

Single-step weakest preconditions allow us to reason about a single instruction at a time. Rule WP-SSWP shows the relation between weakest preconditions and single-step weakest preconditions: informally, it says that (setting aside the antecedent of the separating implication in the postcondition) to reason about a list of instructions, we can reason about the first one, and then the rest. This gives us, for our assembly language, the type of sequential composition we expect from higher-level languages. We can always apply WP-SSWP to transform a goal formulated in terms of weakest precondition into one formulated in terms of single-step weakest precondition, so that we can apply proof rules for individual instructions, and then proceed with the reasoning of the remaining instructions.

Resumption Conditions

We achieve modular reasoning between VMs through resumption conditions, which provide a form of rely-guarantee reasoning tailored for cooperative multitasking between VMs. To ensure that the entire logic integrates with resumption conditions, we bake *RCHolds* into the definition of weakest preconditions, so that we have to prove *RCHolds* when relinquishing control, and in exchange we can assume it when getting control back (as in the postcondition of WP-SSWP). This allows us to write specifications for individual VMs, and prove them separately without having to reason about other VMs' private state, and only having to reason about the

private resources of the current VM and the shared resources that are transferred according to the communication upon scheduling. If a yielding (or scheduling) just happened, we immediately get to assume *RCHolds*, and we can obtain ownerships of the transferred resources stated in the resumption condition by RC-HOLD to continue the reasoning.

Then, to combine the proofs of the local specifications, we have to make sure that the resumption conditions are consistent and compatible, i.e. combined together, they form a unified global protocol, and therefore the combined global specification is valid. To do so, we use the fractional permissions of separation logic [Bornat et al., 2005; Boyland, 2003]: we split the *RC* of a secondary VM in two halves, and let the primary VM and that secondary VM own one half each. Owning half is enough for both VMs, since SS-RUN requires merely half to run the secondary, and RC-HOLD requires merely half to obtain ownership of the resources in the *RC*. In the example above, the protocol is specified by the *RC* of VM1 with the *RC* of VM0 embedded into it. The *RC* of VM1 is split into two fractions owned by the two VMs so that they conform to the same protocol.

Many concurrent separation logics, including Iris, already define a standard mechanism to reason about concurrent programs: invariants. However, resumption conditions are more convenient for the scenarios we consider, as they only require the user to consider interference from other VMs when it occurs, namely at the point of yielding; invariants would force us to consider it (and show that it is not present) at every step of the program. Iris also defines 'non-atomic' invariants, which are a closer fit for our scenarios, as they can group multiple execution steps as a single critical section when holding an exclusive token. However, they do not address the issue completely: a sharing mechanism like invariants is still required to transfer those exclusive token between VMs.

Recursive resumption conditions We have shown in the example above how we can embed one resumption condition into another to construct a run-and-yield protocol between two VMs. In fact, our logic more generally supports recursively defined resumption conditions, which are useful for reasoning about examples where the number of switchings is unknown or unbounded. Consider a 'ping-pong' example, in which a primary VM and a secondary VMi just keep running each other; we can model this protocol as follows:

$$\Psi_i \triangleq \mathsf{R0@0} \overset{\text{reg}}{\longmapsto} \mathsf{Run} * \mathsf{R1@0} \overset{\text{reg}}{\longmapsto} i * \\ \mathsf{RC}_{1/2}\mathsf{@0}\left\{\mathsf{R0@0} \overset{\text{reg}}{\longmapsto} \mathsf{Yield} * \mathsf{R1@0} \overset{\text{reg}}{\longmapsto} i * \mathsf{RC}_{1/2}\mathsf{@}i\,\{\Psi_i\}\right\}$$

The use of *RC* in Ψ_i ensures that Ψ_i is well-defined by the soundness of Iris higher-order ghost states and guarded recursion. Technically, *RC*s are defined using so-called saved propositions, which means that the recursive occurrence of Ψ_i is automatically guarded (even without an explicit 'later' modality $\triangleright$) and hence Ψ_i is well-defined. Using a logic with guarded recursion like Iris means we do not need to be concerned

about soundness of these definitions, as one would have to be if working directly over the operational semantics.

Formalising in Iris

We formalise VMSL using Iris because it allows us to capture and generalize the well-established ideas behind the two logical constructs. We use Iris's primitives and leverage its advanced features, such as higher-order ghost states and guarded recursion, as demonstrated in the recursive example above. The resulting solution is sound and compatible with existing Iris logical constructs thanks to our foundational approach. We use the combination of resumption conditions and invariants in Section 4.4, and believe such compatibility would also be useful to tackle for example interrupts and proper concurrency. Moreover, our solution is language/model-agnostic, therefore can be instantiated with different low-level languages and used to the reasoning of them – e.g., VMSL is obtained by instantiating it with the HVC model.

4.4 Reasoning in the Presence of Unknown VMs

In our full motivating example in Figure 4.1, VM0 runs an unknown VM2 before running VM1 to let it retrieve the shared page. We assume that page pp_2, a page that VM0 and VM1 have no access to, is the only page that VM2 has access to except for its mailbox pages. Since the hypervisor provides isolation between VMs, we would like to show that the effect of VM2 is contained, in the sense that it cannot interfere with the sharing of the page p, nor change its contents. We capture this by showing that the same specification holds for VM0 as in the previous section.

This kind of scenario underpins many use cases of the kind of thin hypervisor we are modelling. For instance, if a secondary VM running some safety-critical service only interacts with the primary VM (running the operating system for scheduling and simple memory sharing), then other VMs cannot manipulate or break the secondary VM through malicious writes to memory.

We leverage the basic memory integrity mechanism of the machine to show *robust safety* for some key scenarios, that is, safety even in the presence of interactions with arbitrary unknown VMs trying to violate memory isolation, including by making hypercalls to attempt to get access to the private memory of other VMs. There are two overall shapes of scenarios:

1. When the primary VM is safe, strong properties hold for the whole system.

2. When the primary VM is compromised, because the primary VM is where the scheduler resides, and because it therefore interacts with all the secondary VMs (at least for scheduling), these strong properties do not hold, but some weaker properties still hold for known secondary VMs.

Proving robust safety Proving robust safety for a machine with only known VMs is straightforward, as the property is captured by VMSL:

1. For each known VM, we prove a weakest precondition.
2. We apply the adequacy theorem, which combines the proved weakest preconditions of all VMs together, to get a valid global execution of the whole machine.

However, this approach does not work directly if an extra unknown VM is considered. To be able to apply the adequacy theorem, we first have to establish a weakest precondition for that unknown VM under conditions that are compatible with the resources used for the other VMs. Because we do not have a concrete program, we do not know whether the program will behave properly, or try to maliciously write to a memory cell that exclusively belongs to another VM, or share memory with other VMs via hypercalls, or any combination of these. Therefore, the questions we face are how to obtain a weakest precondition for an unknown VM, and whether we can use VMSL to establish one.

Inspired by models for capabilities [Devriese et al., 2016; Georges et al., 2024; Swasey et al., 2017], our answer is that we can do so using logical relations. We define two logical relations that are compatible with each other, one for each of the two scenarios. We introduce the logical relation for the first scenario and illustrate it on the example of Figure 4.1 in Section 4.4.1, and describe how the second logical relation is derived by extending the first in Section 4.4.2.

4.4.1 A Logical Relation for Unknown Secondary VMs

To prove examples like Figure 4.1, we define a unary logical relation $\mathcal{R}$ whose fundamental theorem gives us a weakest precondition for any unknown secondary VMi. Our logical relation states that, given the state of the page table and in-flight transactions that determine which memory pages VMi has or may get access to, as defined by InterpAccess, the execution of VMi can be safely resumed, as defined by InterpExecute:

$$\mathcal{R}(i) \triangleq \mathsf{InterpAccess}(i) \mathrel{-\!\!*} \mathsf{InterpExecute}(i)$$

Then, the *fundamental theorem of the logical relation* (FTLR) just states that the logical relation holds for any VMID i except for 0:

$$\forall i.\, i \neq 0 \to \mathcal{R}(i)$$

From the perspective of proving the FTLR, InterpAccess can be regarded as a predicate specifying the exact resources we need to prove the execution of VMi. We define InterpExecute in terms of a *weakest precondition* to capture that if the execution of the VM is resumed, with the resources needed to resume it, then we can execute the VM until it stops or suspends again:

$$\mathsf{InterpExecute}(i) \triangleq \mathsf{RCHolds@}i \mathrel{-\!\!*} \mathsf{wp}\ \texttt{Normal}\ @\ i\ \{\top\}$$

It is sufficient for the postcondition to be $\top$, because we do not need to know what the state of the unknown VM is at the point of halting (in fact, we would not be able to specify it anyway).

Defining InterpAccess

During the execution, VMi may execute any valid instructions, and so we cannot make assumptions about the content of memory of VMi that would restrict its behaviours. Therefore, we have to reason about all possible cases of its execution in the proof of FTLR (which we do by using the proof rules of VMSL).

The definition of InterpAccess for a VMi follows two principles:

1. It must allow us to characterise the behaviour of VMi enough to prove our desired safety property, whatever instructions VMi executes. The way this manifests in the proof is that it must include enough resources for us to be able to apply our proof rules for any instructions.

2. It should not needlessly limit our ability to reason about other VMs. Giving to VMi resources that VMj could own means we might not have necessary resources to prove the specification of VMj. Therefore, InterpAccess(i) should contain just enough resources to reason about VMi.

These two principles make InterpAccess(i) the footprint of running an arbitrary program on VMi. Figure 4.7 shows the top-level definition of InterpAccess.

In general, InterpAccess(i) is parametrised by s_{acc}, the set of pages that VMi has access to, and τ, the map from *Word* to *Transaction* representing all in-flight transactions. Intuitively, the behaviour of VMi, in particular its interactions with other VMs, is (and can only be) restricted by information carried by these two variables. For instance, VMi cannot share a page whose *PageID* is not in s_{acc}, nor retrieve pages shared with another VM according to τ. The main goals of InterpAccess is therefore to interpret these variables with resources, following the two principles above.

Among all the resources of InterpAccess(i), some are exclusively owned by VMi, and some have to be shared between VMi and other VMs due to the communication allowed by HVCs. The shared part is transferred from the primary to VMi upon resumption (via Ψ_i) and is given back to the primary upon yielding (via Ψ_0), using *RC*s. Ψ_i and Ψ_0 are parametrised by an extra τ', to represent new transactions allocated or updated during the suspension of VMi. The connection between τ and τ' is captured by the relation $\tau \sim \tau'$, that is that, the transactions in which VMi is the sender or receiver in τ cannot be touched by other VMs during its suspension, and therefore remain unchanged in τ'. This relation allows us to unify the two, safely replacing τ with τ'. We then only work with τ', which includes all ongoing transactions when VMi is actually executed.

We present this definition by first considering the resources interpreting s_{acc} and τ' as a whole, without distinguishing between exclusively owned and shared, to

$$\begin{aligned}\mathsf{InterpAccess}(i) \triangleq\ & \forall s_{acc}, \tau.\ ❶\mathsf{Pgt}@i \overset{\mathrm{acc}}{\longmapsto} s_{acc} * ❷\mathsf{PgtOea}(s_{oea}) * \\ & ❸\mathsf{MemPages}(s_{oea} \cup \mathit{excl\_pages}(\tau)) * ❹\mathsf{PgtTranP}(\tau) * \\ & ❺\mathsf{RC}_{1/2}@i\ \{\Psi_i\} * \ldots \\ \Psi_i \triangleq\ & \exists \tau'.\ \tau \sim \tau' \wedge ❻\mathsf{TranHandles}(\tau') * ❼\mathsf{PgtTranS}(\tau') * \\ & ❽\mathsf{MemPages}(\mathit{shared\_pages}(\tau')) * ❾\mathsf{RC}_{1/2}@0\ \{\Psi_0\} * \ldots\end{aligned}$$

Figure 4.7: The shape of the definition of $\mathsf{InterpAccess}(i)$. All predicates are implicitly parametrised by i if i is mentioned in their definitions. We refer readers to the Coq formalisation for the full definition.

argue why the unknown VM needs them, and later argue why and how to divide them into owned and shared portions.

Intepreting s_{acc}

The interpretation of s_{acc} is split as follows: First, ❶ states that these pages are accessible to VMi, which is required by all the proof rules (e.g. ❸ of SS-MOV). Second, ❷ provides page table resources for pages that VMi owns and has exclusive access to (denoted as s_{oea} and computed from s_{acc} and τ), which is defined as $\mathbin{\ast}_{p \in s_{oea}} \mathsf{Pgt}@p \overset{\mathrm{own}}{\longmapsto} i * \mathsf{Pgt}@p \overset{\mathrm{excl}}{\longmapsto} \mathsf{True}$ (or $\mathsf{PgtOE}(s_{oea}, i, \mathsf{True})$ in short). Those resources are required by the proof rules (e.g. ❻ of SS-SHARE) if VMi shares pages that are in s_{oea}.

These two components are exclusively owned by VMi since no other VMs may require them. Another necessary but partially shared component is the memory of s_{acc}, $\mathsf{MemPages}(s_{acc})$, which is required by rules for memory access instructions. We divide s_{acc} (and the predicate correspondingly) in two parts: memory pages that VMi has exclusive access to, and the remainder that is shared with other VMs. The former is captured by s_{oea} plus pages that are lent to VMi, collected by $\mathit{excl\_pages}(\tau')$, as in ❸; the latter is collected by $\mathit{shared\_pages}(\tau')$ as in ❼.

Interpreting τ'

In general, three kinds of resources could be necessary to allow VMi to perform memory sharing HVCs on t: $\mathsf{Tran}@h \overset{\mathrm{tran}}{\longmapsto} t.\mathsf{meta}$ is necessary to refer to t for any sharing HVCs; $\mathsf{Tran}@h \overset{\mathrm{rtrv}}{\longmapsto} t.\mathsf{retri}$ is necessary to retrieve the access to shared pages $t.\mathsf{pgs}$; and $\mathsf{PgtOE}(t.\mathsf{pgs}, \_, \_)$ is necessary to update the status of the shared pages.

These resources are split into fractions such that some are owned by VMi, and some are shared. The owned and shared fractions are used to interpret transactions of τ and τ' respectively, and unified later by $\tau \sim \tau'$ (so they both interpret τ'). For instance, a points-to for transactions is split into three fractions that must agree on their values. One third in some cases is owned by VMi, and at least another one third is shared in all cases. The points-tos for the page table are split and unified in the

Table 4.2: Select cases of how a transaction t is interpreted. Column one gives metadata and state of t, where j and k are VMIDs of two other VMs. Columns two to four give the required fractions of the three kinds of required resources. $1/3 + 2/3$ under column two means $\mathsf{Tran@}h \overset{\mathsf{tran}}{\longmapsto}_1 t.\mathsf{meta}$ is required in total, with $1/3$ of it owned by the unknown VMi, and $2/3$ shared.

sndr, rcvr, type, retri	$\mathsf{Tran@}h \overset{\mathsf{tran}}{\longmapsto} \mathsf{meta}$	$\mathsf{Tran@}h \overset{\mathsf{rtrv}}{\longmapsto} \mathsf{meta}$	$\mathsf{PgtOE}(\mathsf{pgs}, \_, \_)$
i, j, Share, False	$1/3 + 2/3$	1	$1/3 + 2/3$
i, j, Donate, False	1	1	1
j, i, Share, True	$2/3$	$1/2 + 1/2$	$2/3$
j, i, Lend, True	$2/3$	$1/2 + 1/2$	$2/3$
j, k, _, _	$1/3$	0	$2/3$

same way, and the splitting is then lifted to PgtOE. At least two fractions of PgtOE that interpret t are shared, which allows us to derive the fact that pages shared by two transactions are disjoint by leveraging the exclusivity of $\mathsf{PgtOE}_{2/3}$ that is derived from that of the underlying page table points-tos.

Now let us zoom in on several representative cases outlined in Table 4.2 to see why those resources are distributed like this. In case "i, j, Share, False", VMi is the sender, and therefore the owner of the shared pages. All fractions of the three resources are required as the sender could Reclaim access, recycling the two transaction points-tos and updating PgtOE by the proof rule. The owned fractions allow VMi to remember that it has shared t.pgs even after a suspension. The receiver doesn't need them to Retrieve or Relinquish. In case "i, j, Donate, False", all resources are shared, as the receiver could Retrieve, which gives it ownership of the pages t.pgs. In case "j, i, Lend, True", VMi as the receiver does not own page table resources nor the points-tos for transaction, as there is no way for it to get ownership of those pages (and full ownership of the three resources is not required by the proof rules of Retrieve or Relinquish). However, it owns half of the retrieval points-to, so that it can remember the fact that it has retrieved after a suspension. In the last case "j, k, _, _", VMi is neither the sender nor the receiver (which is the case of VM2 in our example), only the minimum amount of resources is required (in our example, $\mathsf{Tran@}h \overset{\mathsf{tran}}{\longmapsto}_{1/3} (0, 1, \{p\}, \mathsf{Share})$ and $\mathsf{Pgt@}p \overset{\mathsf{own}}{\longmapsto} 0 * \mathsf{Pgt@}p \overset{\mathsf{excl}}{\longmapsto} \mathsf{False}$).

Resources specified in Table 4.2 are distributed in ❹, ❻, and ❼. ❻ includes the least amount of fractions required by all cases, i.e. $1/3$, 0, and $2/3$, of the three kinds of resources respectively, for each transaction in τ':

$$\mathop{\Large *}_{h \mapsto t \in \tau'} \mathsf{Tran@}h \overset{\mathsf{tran}}{\longmapsto}_{1/3} t.\mathsf{meta} * \mathsf{PgtOE}_{2/3}(t.\mathsf{pgs}, t.\mathsf{sndr}, (t.\mathsf{type} =? \mathsf{Share}))$$

Remaining owned and shared fractions are distributed in ❹ and ❼ respectively with definitions of similar shapes as ❻.

General Protocols

❾ in Figure 4.7 is one half of the resumption condition specifying which resources are supposed to be returned back to the primary VM to resume its execution. Generally speaking, the same resources transferred to VMi are passed back, plus the recursive resumption condition of VMi which allows the primary to run VMi multiple times.

$$\Psi_0 \triangleq \exists \tau.\, \mathsf{TranHandles}(\tau) * \mathsf{PgtTranS}(\tau) * \mathsf{MemPages}(\mathit{shared\_pages}(\tau)) * \\ \ldots * \mathsf{RC}_{1/2}@i\,\{\Psi_i\}$$

We call such a protocol specified by the two resumption conditions the *general protocol* of VMi. It is general in the sense that it specifies necessary resources to support arbitrary execution of VMi, for arbitrary numbers of resumptions, and it is used to reason about unknown VMs. In the case where the primary VM is unknown, we sometimes need an additional mechanism for reasoning about sharing between communicating VMs, see the example considered in Section 4.4.2.

Proving the FTLR

To show that the FTLR holds, we have to consider all possible instructions since the program of the VM is unknown. For each instruction, we apply the corresponding general proof rule of VMSL. See the Coq formalisation for the proof.

Instantiating the FTLR

We now demonstrate how we use the logical relation to reason about the full motivating example by instantiating the FTLR. Recall that our approach is to

1. show a weakest precondition for each of the three VMs, assuming resources describing the initial state of the machine; and
2. combine them to apply the adequacy theorem, which provides these resources.

The weakest precondition for VM1 can be proved as for the simplified example. To show the weakest precondition for VM2, we instantiate the FTLR with *VMID* 2. We then have to pick proper s_{acc} and τ such that the required resources are disjoint and consistent with resources required by the other two known VMs. That is, all initial resources are exclusively owned by one VM, and the protocols specified in resumption conditions agree with each other. We let τ be $\emptyset$, since at the beginning there are no transactions, and we let s_{acc} be $\{p_{tx2}; p_{rx2}; pp_2\}$. To show the weakest precondition for VM0, which now runs VM2 before VM1, we have to show the resumption condition of VM2 specified in $\mathsf{InterpAccess}(2)$. In particular, we let τ' be $\{h \mapsto (0, 1, \{p\}, \mathsf{Share}, \mathsf{False})\}$, whose interpretation in TranHandles will disallow any malicious HVCs, such as retrieving access to p, by VM2. The same resources are included in Ψ_0 and given back, so this transfer does not affect the reasoning about the two known VMs after running VM2.

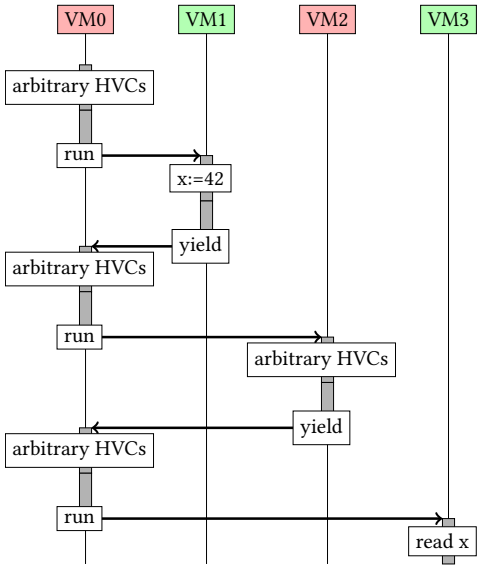


Figure 4.8: A compromised primary VM is also contained: memory integrity (illustrating defensive code).
This assumes VM1 and VM3 initially exclusively share a page p containing location x.

Capturing Safety

The fact that we are able to prove (using our logical relation) that VM0 and VM1 can safely share a page, even though VM2 runs in between and gets the opportunity to try to interfere, shows that our underlying machine-with-HVCs model is *secure*, in the sense that executing those HVCs will not break isolation unintentionally.

4.4.2 A Logical Relation for Unknown Primary VMs

We have shown how to reason in the presence of unknown secondary VMs using our first logical relation. However, secondary VMs also get some guarantees when the primary VM is unknown (and possibly compromised). For example, consider the scenario in Figure 4.8: only two secondary VMs, VM1 and VM3, are known, and a page p with 42 stored in it is shared between them. We would like to show that VM3 can read that same value from the page, even with the unknown primary VM0 in addition to the unknown secondary VM2. In this example, as before, we can instantiate the FTLR to get a weakest precondition for VM2, but we cannot do the same for VM0.

To deal with scenarios with an unknown primary VM, we develop a second logical relation, whose FTLR gives a weakest precondition for the primary VM. We ensure that this second logical relation is also compatible with our previous logical relation. This enables us to show safety of scenarios with both arbitrary unknown primary and secondary VMs, including the example above. In such scenarios, programs of

known secondaries have to be written defensively, as they may be scheduled at any point. In this section, we show how we design and use this second logical relation, and refer the reader to the Coq formalisation for the full definition.

The statement of the FTLR of the new logical relation is symmetrical to the previous one: we now require i to be 0. As before, InterpExecute is defined as just wp 0 @ `Normal` {⊤}, and moreover *RCHolds* is not needed as we always run the primary first. The difference is in InterpAccess, which generalises the former to support running arbitrary secondary VMs, namely the extra power of the primary VM. From the perspective of resources, the new InterpAccess includes

1. resources that supports VM0's execution except for running other VMs, which is identical to what is required by a secondary VM as in Section 4.4.1; and

2. resources required by resumption conditions of all secondary VMs to support running these VMs, which is basically their resumption conditions plus the union of resources required by them.

The crux of defining the new InterpAccess is specifying *all* the resumption conditions, i.e. protocols between all secondaries and the primary. For unknown secondaries, as shown in the previous subsection, we can use the general protocol. For known secondaries, because we want our FTLR to be generic in their code, the protocol cannot depend on their code (so, here, we cannot take the approach we used for the example in Figure 4.1). Moreover, we cannot use the general protocol for known code either, as it is too general to be used to prove e.g. the example in Figure 4.8. The technical problem arises from: (1) the very loose assumption on the content of memory, which is quantified over existentially in the general protocol. That is, we want to show the shared page p contains a specific number, but the general protocol only gives us that there is some number in p. (2) the fact that resumption conditions only allow transferring resources along the scheduling control flow via the primary VM (as illustrated on the left of Figure 4.9). With the cooperative scheduling mechanism we model, secondary VMs can only yield to the primary VM, not directly from one secondary VM to another. This means that in this example, the shared page p can only be transferred between VM1 and VM3 with VM0 as a middleperson.

Our Approach

Instead, we exclude the page p from the general protocol, and share it between VM1 and VM3 in another way (which we can do since p is not accessible to VM0). To do this, we use invariants as a complementary resource sharing mechanism, for resources that cannot or should not be shared via the general protocol. In this example, assuming p's value is always 42 after VM1 writes to it, we can establish a trivial invariant, as illustrated in Figure 4.9, with the memory resources of page p.

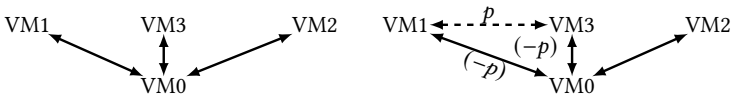


Figure 4.9: An illustration of how resources are shared among VMs in Figure 4.8. Regular arrows represent the resources of the general protocol, where $(-p)$ means the resources of page p are excluded. Instead, those resources are shared via an invariant represented as the dashed arrow.

How We Implement Our Approach

Recall that the general protocol specifies the resources a secondary shares with all other VMs, although they are only ever transferred via the primary. It indicates that it is safe to run an unknown primary without resources that secondaries shared with other secondaries in the general protocol. We therefore can divide the resources of the general protocol into *slices*, one for each pair of VMIDs, which only contain one-to-one shared resources. This way, we can now safely remove secondary-to-secondary slices from the general protocol between a secondary and the primary. We then parametrise the logical relation by the secondary-to-secondary slices, thereby allowing the user of the FTLR to decide which of those slices are (partially) transferred via the unknown primary. For instance, resources that VM1 shares using its general protocol are divided into three slices containing resources that it shares with (1) VM0; (2) VM2; and (3) VM3. We say the slice from VM1 to VM2 is *full* if it contains all related resources required by the general protocol between VM1 and the primary. We then instantiate the FTLR with full slices (1) and (2), and (3) minus the memory of page p, to exclude that page from the VM1-to-VM3 slice. By doing so, yielding of VM1 will not require the resources for page p, and therefore we can use it to establish the invariant. Moreover, by letting slices from VM2 to other VMs be full, we can actually recover the general protocol of VM2, therefore making the two logical relations compatible.

4.5 Related Work

Hypervisor and OS verification There are several lines of work on hypervisor verification, including HASPOC [Baumann et al., 2016, 2019], SeKVM [DBL, 2021; Li et al., 2021; Tao et al., 2021], Hyper-V [Leinenbach and Santen, 2009], and seL4 [Klein et al., 2014, 2009].

The HASPOC project is aimed at designing a secure virtualisation platform for ARMv8, for which they prove information-flow security. They introduce an idealised model in which information-flow security holds by construction, and prove a bisimulation between it and the concrete platform model. In their model, each VM's memory is isolated and cannot be shared; instead, inter-VM communication is restricted to a messaging mechanism similar to the one we model.

The main focus of SeKVM is on hypervisor verification. As part of it, they

capture generic isolation properties between virtual machines and their hypervisor (based on KVM) in the form of non-interference results about their combined model of the machine and the hypervisor, capturing both integrity and secrecy. They support memory sharing in a much more restrictive way, only allowing a VM to share encrypted data with the less privileged portion of the hypervisor to support I/O virtualization.

Microsoft's Hyper-V is an industrial hypervisor partially verified with the VCC verification suite [Cohen et al., 2009], and their verification effort focuses on low-level concurrent C code. Most of their verification effort relates the hypervisor implementation to its specification, but not on validating that top-level specification, nor on its security properties.

seL4 is a formally verified OS kernel. Whereas in our setting, scheduling is outsourced to a primary VM, in their setting, scheduling is done by seL4 itself. In addition to functional correctness, seL4 includes a proof of some non-interference properties [Murray et al., 2013], which they prove over the kernel specification. The integrity result for seL4 [Sewell et al., 2011] considers a small operating system that manages a set of capabilities with various authorities (write, read, send, receive, grant, etc.) over various objects. Their operating system corresponds to the combination of our hypervisor and a "receptive" primary that waits for requests, checks they are allowed, and executes them. In that setting, they consider what kind of capabilities are accessible through privilege escalation. This is similar to the way in which our logical relations have to consider what can be acquired transitively through transactions and memory.

These efforts primarily focus on verifying the implementation of system software (including APIs exposed to clients). Our work is complementary, in that our approach factors the integrity (but not the secrecy) part of their security results into a logic to reason about concrete programs using hypercall APIs, and a logical relation that captures isolation. This, in contrast to their approaches, enables us to give specifications and verify individual concrete scenarios, whereas, in our terms, their results are concerned with composing exclusively unknown VMs.

In addition, these lines of work make drastic simplifying assumptions, as the actual behaviour of page tables, especially in the presence of concurrency, is only beginning to be understood precisely enough for verification [Simner et al., 2022]. Nonetheless, there is some work on hypervisor verification against authoritative models: Nienhuis et al. [Nienhuis et al., 2020] and Bauereiss et al. [Bauereiss et al., 2022] prove security properties above full-scale, authoritative, formal ISA models of the CHERI and Morello capability architectures. These properties are finer-grained than ours thanks to capabilities, but weaker in that they are architectural invariants, and thus cannot rely on properties of known code. Sammler et al. [Sammler et al., 2022] develop a separation logic above authoritative, formal ISA models of Arm-A and RISC-V by specialising the ISA definition to partially concrete opcodes through (unverified) symbolic evaluation [Armstrong et al., 2021]. They focus on verifying local specifications of known code, including some exception handlers.

Reasoning about low-level code The details of low-level code make it a natural target for mechanisation, and there is extensive work on the topic. Our work follows in the footsteps of the CAP [Feng and Shao, 2005; Ni and Shao, 2006; Ni et al., 2007; Yu and Shao, 2004] family of Hoare logics for low-level code, which tackle for example code pointers and cooperative multitasking (which we return to later). In mechanising their logics directly in Coq, without an intermediate logic like Iris, they identify challenges concerning higher-order code (via code pointers), separation, rely-guarantee reasoning, etc., and also note opportunities offered by mechanisation, for example 'open' proof rules that are defined as lemmas over the operational semantics rather than hard-coded into the logic. Concurrently with the CAP work, mechanised variants of separation logic have long been used to reason about assembly code [Cai et al., 2007; Jensen et al., 2013; Kennedy et al., 2013; Myreen and Gordon, 2007]. Iris generalises this approach, building on separation logic to encapsulate the logical constructions that are helpful to reason about programming languages in a language-independent way. Our work (like Georges et al. [Georges et al., 2021]) demonstrates how such a rich logic does indeed make it tractable to tackle many of the challenges of low-level code identified by the CAP line of work.

Single-step weakest preconditions Decomposing reasoning about a sequence of instructions into reasoning about each instruction one by one is quite intuitive, but often raises proof engineering challenges, and some solutions are 'folklore'. For example, Erbsen et al. [Erbsen et al., 2021, §4.3] capture individual steps, and compose them with an 'eventually' operator similar to a transitive closure. Our single-step weakest precondition, like the standard Iris weakest precondition, is defined purely in terms of the type of operational semantics that Iris takes as input, and thus factors out this aspect of instantiating Iris for low-level code, independently of the language. For example, we believe that our approach could be used by the capability machine formalisation of Georges et al. [Georges et al., 2021] to simplify some of their proof engineering.

Cooperative multitasking and resumption conditions Programming over the fragment of the FF-A hypercall API we consider, where secondary VMs run until they explicitly yield to the primary VM, is effectively a form of cooperative multitasking with a programmed scheduler. Again, we follow in the footstep of the CAP line of work [Feng and Shao, 2005; Yu and Shao, 2004], but benefit from a modern, mechanised separation logic. Moreover, in our terms, the CAP setting corresponds to only composing known secondary VMs sharing some pages with a primary that merely schedules secondary VMs. Using our logical relations which capture bounds on the effect of arbitrary code, we go further, and capture the composition of known and unknown code.

Capability machines Capabilities [Arm, 2021; Carter et al., 1994; Watson et al., 2019; Wilkes and Needham, 1979] are an alternative hardware mechanism for access

control, in the form of dynamically checked unforgeable tokens of authority, typically granting some type of finer-grained access to a portion of memory. Proofs of safety for capability machines have also used unary, unityped logical relations, e.g. [Georges et al., 2024, 2022; Skorstengaard et al., 2019]. However, these logical relations are quite different from ours, because of the different underlying mechanisms. Their logical relation involves recursion through the heap, as a capability can give access a portion of the heap which gives access to further capabilities; whereas in our setting, there is a clear stratification of page tables 'above' the memory accessible to VMs. Because we do not have this recursion, a VM does not need to hand over all of its memory to a global invariant, and instead can locally keep the resources for the memory that it does not share, which leads to more direct reasoning at the expense of some complexity in the definition of our logical relation.

4.6 Conclusion

We have formalised a substantial fragment of Arm's FF-A ABIs as an operational semantics in which HVCs are primitive steps and we have demonstrated that the model is secure, in the sense that VMs running unknown and possibly malicious code cannot break isolation unintentionally. In more detail, we have developed VMSL, a novel separation logic for modular reasoning about known VMs communicating above FF-A. In particular, VMSL supports 'VM-local' reasoning via its notion of *resumption conditions*, which capture interaction between VMs and thereby reduces reasoning about their interaction to sequential reasoning. Moreover, we have shown how to use the logic to develop logical relations that capture the intended isolation guarantees and which can be used to formally prove robust safety for communicating known VMs that interact with VMs running unknown code. Finally, we have applied these to prove security in key scenarios that capture the typical interaction cases between VMs with various trust relations.

Future work includes extending our model with concurrency and non-cooperative scheduling. We are also interested in adapting our model to the pKVM [Deacon, 2020; Google LLC, 2021; Perret, 2020] ABIs, which is different from the FF-A ABIs but similar in spirit. It would also be interesting to show that an implementation of a hypervisor is a formal refinement of (a more detailed version of) our model.

5

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

Abstract

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.

5.1 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/Rocq, using the Iris program logic framework [Jung et al., 2018b, 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 Figure 5.1. 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., 2015a] 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.

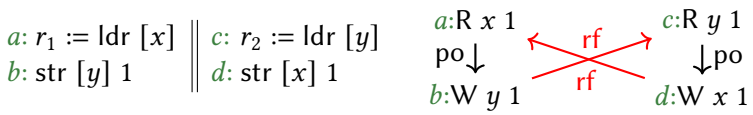


Figure 5.1: LB+pos

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 or-

dering (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.

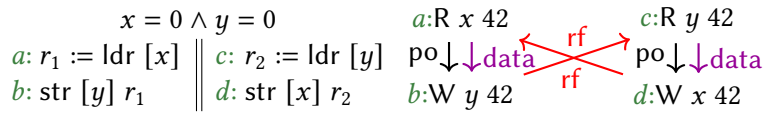


Figure 5.2: LB+datas

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 Figure 5.2 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/Rocq, 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 Section 5.2. We explain the key ideas of our logic informally in Section 5.3. In Section 5.4, 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 Section 5.5, 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 Section 5.6, we define the model of AxSL in Iris, following the same three stages, and present our non-standard definition of weakest precondition. In Section 5.7, we present our non-standard proof of adequacy of AxSL in Iris. In Section 5.8, we discuss some technical aspects and limitations of our work. We discuss related work in Section 5.9, and how our work can be used and extended further in Section 5.10. 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 Section 5.4.3.
- We explain the novel ideas of AxSL^{Arm} in that simpler setting, building two logics for that simple SC language: AxSL^{SC} and $\text{AxSL}^{\text{SCExt}}$. These two simpler logics work as explanatory steps when building up the syntax and the semantic model of the AxSL^{Arm}.
  - We show how to define a first straightforward logic, AxSL^{SC}, on top of our novel 'opax' style of semantics in Section 5.5.2, and how to define a semantic model for it in Section 5.6.3.
  - We then show how to define a second, more elaborate logic, $\text{AxSL}^{\text{SCExt}}$, that uses the ideas that make AxSL^{Arm} work in the setting of relaxed memory, but still in the simple setting of sequential consistency in Section 5.5.3, and present its semantic model in Section 5.6.4.
- We give a self-contained presentation of the definitions of the model of AxSL^{Arm}, using precise definitions that were omitted in the original paper because of space constraints, in Section 5.6.5.
- We expand the explanation of various technical definitions and proofs, and add illustrations to make the technical material more accessible.

[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 refine the model of AxSL$^{\text{Arm}}$, leading to some technical improvements that we elaborate on in Section 5.8.1, in particular a better proof of adequacy.

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

5.2 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, 2007; O'Hearn, 2007], 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 Section 5.3.2.

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

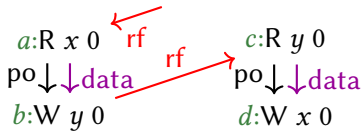


Figure 5.3: An SC execution of LB+datas

defines compound relations derived from these, e.g. for Arm-A — which we will use as our main case study — the model of Figure 5.4 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 Figure 5.1 is allowed by the Arm-A axiomatic model because the plain po relation, between reads and writes to different addresses, is not included in the ordered-before ob that is required to be acyclic (or in the internal or atomicity requirements). The candidate execution for LB+data at the bottom of Figure 5.2 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 Figure 5.3 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 (Figure 5.5).

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

FSL generalises RSL to reason about C11's release and acquire fences, but its assertions are still persistently freely usable along po, so they have to choose between

```
(* 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
```

Figure 5.4: 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.

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 Section 5.9.

5.3 Key Ideas

5.3.1 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

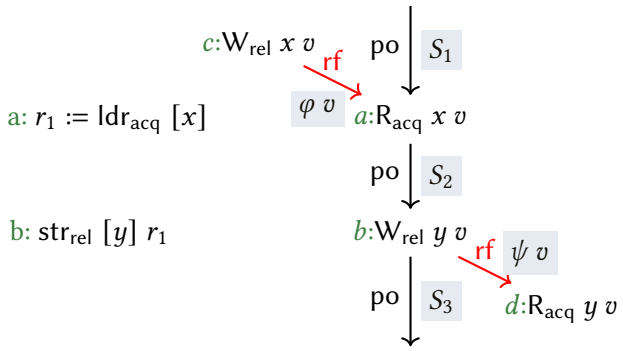


Figure 5.5: Thread-local part of a candidate execution, annotated with the logical resources flowing on edges. The thread program is on the left. Resources $S_{1,2,3}$ are those in hand at each program point, and the protocol specifies the resources $\varphi\ v$ and $\psi\ v$ passed along the release-acquire edges for x and y.

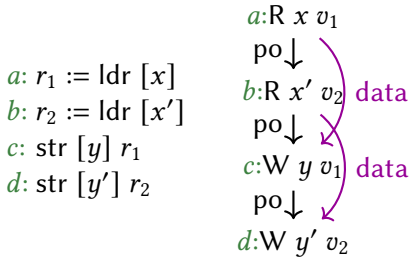


Figure 5.6: Intra-thread concurrency

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 Figure 5.6. The data dependencies order a with c, and b with d, but that is all the ordering we get. In particular, nothing orders the last read b, before the first write c – in contrast to the release/acquire case.

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

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

$$(r \overset{\mathrm{r}}{\mapsto} v@\{a\}) * (a \rightsquigarrow (\varphi\ v))$$

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 $a \rightsquigarrow (\varphi\ v)$ records the resources gained (according to protocol φ) from the load, *tying* them to its event ID a. (The $\varphi, \psi, \ldots$ 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 $r \overset{\mathrm{r}}{\mapsto} v\&(\varphi\ v)$, 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., 2015b] 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., 2018b, 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 Sections 5.3.2 and 5.4.3), 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 Section 5.5.6). 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 $a \rightsquigarrow P$ assertion is itself defined using ghost state, using the fact that Iris ghost state is higher-order, in the sense that it is mutually defined with Iris propositions.

Using these assertions, we can write concrete proofs for synchronisation involving thread-local dependencies, as sketched in Figure 5.7, without relying on the

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

Figure 5.7: Proof sketch for a plain-access (non-release/acquire) version of Figure 5.5. The flow implication is on line 4.

program-order strength of release-acquire reasoning we illustrated in Figure 5.5. (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 Section 5.5.) 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 $a \rightsquigarrow (\varphi\ v_1)$. 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 $\varphi\ v_1$ 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 $(R_0 * \varphi\ v_1) \Rrightarrow (\psi\ v_1)$.

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 Figure 5.1, 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 $\varphi\ v_1$ 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 Figure 5.6, 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 $a \rightsquigarrow P$, we can split $a \rightsquigarrow (P * Q)$ into $(a \rightsquigarrow P) * (a \rightsquigarrow Q)$ and frame the latter off, as in Figure 5.8, see Section 5.6.5.

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

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

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

$\{r \overset{\mathrm{r}}{\mapsto} v@\{a\} * a \hookrightarrow P\}$

b: $\mathsf{str}\ [y]\ r$ // 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: $\mathsf{str}\ [z]\ r$ // uses Q

$\{r \overset{\mathrm{r}}{\mapsto} v@\{a\} * a \hookrightarrow \top\}$

Figure 5.8: Splitting tied resources

5.3.2 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., 2018b, 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 $\mathsf{po} \cup \mathsf{rf}$ 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 (Section 5.4), 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 Section 5.5).

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

[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.

whole-thread bounds for barriers), abstracting the set of source events for each. These integer timestamps might be technically easier to work with than graphs, but we found the explicit nodes with explicit edges of the axiomatic model helpful in developing our model of assertions. In a sense, our opax semantics is a reformulation of an axiomatic model made to look more like a promising model, but with the advantage that changes to the axiomatic model apply directly.

Note that, while our opax 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 (Section 5.6.5) and a non-standard proof of adequacy (Section 5.7) (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.

5.3.3 The Third Problem: Structuring the Adequacy Proof

Finally, given a proof in AxSL^{Arm} using our new assertions, we then need an adequacy theorem (Section 5.7), 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 $\mathsf{po} \cup \mathsf{rf}$, which prevents doing an induction on it. **Our third key idea** is that, to solve this tension, we can split the proof of adequacy in two phases: first along po, and then along ob. The first phase, along po, uses thread-local resources to establish, for each thread, and for each memory event of that thread, that the flow implication for that event holds. The second phase, along ob, stitches the flow implications together. For example, this second phase walks through the thread of Figure 5.6 twice, for the two disjoint components of ob: once from a to c, and once from b to d (with both orderings being possible).

5.4 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] (Section 5.4.1), 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/Rocq 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 Figure 5.9. 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, Figure 5.9 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 Section 5.5 build upon. TinySc is syntactically a sublanguage of 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.

5.4.1 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.

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 $\mathrm{Next}_T : \mathrm{Outcome}\ T \to (T \to \mathrm{IMon}\ A) \to \mathrm{IMon}\ A$ and $\mathrm{Ret} : A \to \mathrm{IMon}\ A$. Rather

$$
\begin{array}{lll}
i_{\mathsf{TinyArm}} ::= & & \text{instructions} \\
\mathsf{nop} & & \\
\mid r := t & & \text{register assignment} \\
\mid \mathsf{br}\ x & & \text{branch to address } x \\
\mid \mathsf{bne}\ t\ x & & \text{conditional branch} \\
\mid r := \mathsf{ldr}_{os,vr}\ [t_{\mathsf{addr}}] & & \text{memory load} \\
\mid r := \mathsf{str}_{os,vr}\ [t_{\mathsf{addr}}]\ t_{\mathsf{data}} & & \text{memory store} \\
\mid \mathsf{dmb\ sy} \mid \mathsf{dmb\ st} \mid \mathsf{dmb\ ld} \mid \mathsf{isb} & & \text{Arm-A fences}
\end{array}
$$

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

$$
\begin{array}{lcl}
r \in \mathrm{Reg} \triangleq \{r_0, r_1, \ldots\} & & \\
v, x \in \mathrm{Word} \triangleq 0..2^{64} - 1 & & \\
op & ::= & + \mid - \mid \times \\
os & ::= & \mathsf{plain} \mid \mathsf{relacq} \mid \mathsf{weakacq} \\
t & ::= & v \mid r \mid t_1\ op\ t_2 \\
vr & ::= & \mathsf{nexcl} \mid \mathsf{excl}
\end{array}
$$

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

Figure 5.9: Instructions and syntactic sugar of TinyArm and TinySc

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 Figure 5.10. A plain, non-exclusive load $r := \mathsf{ldr}\ [t_{\mathsf{addr}}]$, 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 Section 5.4.3 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).

A non-exclusive store release $\mathsf{stlr}\ [t_{\mathsf{addr}}]\ t_{\mathsf{data}}$, at t_{addr} of t_{data}, elaborates into the elaboration of the evaluation of terms t_{addr} and t_{data} to some x and v, using the auxiliary function $\mathbb{T}[\![-]\!]$, followed by a release (and non-exclusive) memory write to x of v with address data dependencies and again a PC increment (the dummy register

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

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

Figure 5.10: 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.

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.

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 $\mathbb{D}[\![-]\!]$ 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.

5.4.2 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.

5.4.3 Our Opax Concurrency Model Semantics

As discussed in Section 5.3.2, 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.

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 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 Figure 5.4.

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, $s_{\mathsf{init}}(c_{tid})$ 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:

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

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.

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 Figure 5.11. 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.

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

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

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

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

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

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

Figure 5.11: Selected reduction rules of opax semantics for TinySc

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.)

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 Figure 5.12 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 Figure 5.10 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 Figure 5.10 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.

A-MEM-READ-NEXCL

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

A-REG-WRITE

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

Figure 5.12: 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 yellow.

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 of weakest preconditions. Second, the guessing does not appear in the definition of weakest preconditions, which takes the guessed graph as a parameter; instead, the guessing is handled by a

quantification in the adequacy theorem, when the proofs of the individual threads are combined.

Infinite Executions

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

5.5 The Logics

The goal of this section is to build up to AxSL^{Arm}. We do this incrementally, introducing two intermediate logics to explain the different building blocks of AxSL^{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^{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^{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: $\text{AxSL}^{\text{SCExt}}$, a logic with both foundations and an assertion style compatible with relaxed concurrency. For simplicity, $\text{AxSL}^{\text{SCExt}}$ stays in the context of sequential consistency, but follows the fine-grained resource management style sketched in Section 5.2. In particular, $\text{AxSL}^{\text{SCExt}}$ employs 'tied-to' assertions and flow implications in order to be compatible with relaxed concurrency.

(3) Finally, we present AxSL^{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 Section 5.6.5, while the high-level rules explicitly used in proofs of programs are for the surface instructions of Figure 5.9, derived from the former by reasoning about instruction elaboration (Section 5.4.1). Before diving into the three logics, we first discuss the resource transfer mechanism that they employ.

5.5.1 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 Section 5.5.7). 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 Section 5.9. 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.

5.5.2 Dealing with Opax Semantics: The AxSL^{SC} Logic

AxSL^{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 Section 5.8.2). 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.

Proof Rules for Microinstructions

SC-HT-MICRO-MEMREAD

$$\left\{ \blacksquare\text{PoPred}(e_{\text{po}}) * \forall e, v, e_w. \left(\begin{array}{c} \blacksquare\text{GraphFactsR}(e, x, v, e_w, e_{\text{po}}) * \blacksquare\Phi(x, v, e_w) \\ \Rrightarrow \blacksquare P(e, x, v, e_w, e_{\text{po}}) * \Phi(x, v, e_w) \end{array} \right) \right\}$$

$$\text{MemRead } x$$

$$\left\{ v.\, \exists e, e_w.\, \blacksquare\text{PoPred}(e) * \text{GraphFactsR}(e, x, v, e_w, e_{\text{po}}) * P(e, x, v, e_w, e_{\text{po}}) \right\}_{tid,\Phi}$$

SC-HT-MICRO-MEMWRITE

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

$$\text{MemWrite } x\ v$$

$$\left\{ ().\, \exists e.\, \blacksquare\text{PoPred}(e) * \text{GraphFactsW}(e, x, v, e_{\text{po}}) \right\}_{tid,\Phi}$$

SC-HT-MICRO-REGWRITE

$$\{r \overset{\text{r}}{\mapsto} \_\}\ \text{RegWrite } r\ v\ \{().\, r \overset{\text{r}}{\mapsto} v\}_{tid,\Phi}$$

Figure 5.13: Seleted proof rules of AxSL$^{\text{SC}}$

Figure 5.13 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

$$\{P\}\ \text{MemRead } x\ \{v.\, Q\}_{tid,\Phi}$$

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 $e{:}\mathsf{R}\ x\ v$ 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 postcondition, 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 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 $\mathrm{AxSL}^{\mathrm{SCExt}}$, 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 $T.\mathit{regs}$ in the opax rule A-REG-WRITE.

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

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

Figure 5.14: 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, ...).

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., 2023b]. We use Hoare triples for presentation, but use weakest preconditions in our formalisation. Our Hoare triples for a single instruction i are of the form $\{P\}\ a{:}\ i\ \{a'{:}\ Q\}_{tid,\Phi}$, 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 $\mathsf{pc} \overset{\mathsf{r}}{\mapsto} a$, and Q is combined with $\mathsf{pc} \overset{\mathsf{r}}{\mapsto} a'$ 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 5.14) 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 Figure 5.15 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 with immediate operators by SC-HT-MICRO-MEMWRITE. We use these two instruction rules in the message passing example below.

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$^{\mathsf{SC}}$ on a

SC-HT-INST-LDR

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

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

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

SC-HT-INST-STR

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

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

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

Figure 5.15: Two instruction proof rules with instruction triples. $a + 4$ in the post-condition indicates the address of the next instruciton.

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

Figure 5.16: MP in SC: $r_1 = 1 \longrightarrow r_2 = 42$

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.

Specification We want to be able to prove that if the load of the flag reads 1, then the load of the data will read 42; formally, $r_1 \overset{\mathrm{r}}{\mapsto} v * r_2 \overset{\mathrm{r}}{\mapsto} v' * (v = 1 \longrightarrow v' = 42)$.

Picking the protocol The proof sketch of Figure 5.17 relies on extensive graph reasoning. We first specify the protocol used in the proof. For the data address, we pick $\Phi(data, v, e) \triangleq \mathrm{Initial}(e) \vee v = 42$, where $\mathrm{Initial}(e)$ denotes that the event is an initial write that necessarily has value 0. It is not possible to require that v be 42 in all cases, because the initial write would then not satisfy the protocol. For the flag address, we pick

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

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:

1 $\{\mathsf{PoPred}(-)\}$
2 $\mathsf{GraphFactsW}(a, \mathit{data}, 42, -)$
$\Rrightarrow$
3 $\Phi(\mathit{data}, 42, a) * a{:}\mathsf{W}\ \mathit{data}\ 42$
4 $a{:}\ \mathsf{str}\ [\mathit{data}]\ 42$
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 $b{:}\ \mathsf{str}\ [\mathit{flag}]\ 1$
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 $c{:}\ r_1 := \mathsf{ldr}\ [\mathit{flag}]$
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 $d{:}\ r_2 := \mathsf{ldr}\ [\mathit{data}]$
18 $\{\mathsf{PoPred}(d) * \exists v, v'.\ r_1 \overset{\mathsf{r}}{\mapsto} v * r_2 \overset{\mathsf{r}}{\mapsto} v' * (v = 1 \longrightarrow v' = 42)\}$

Figure 5.17: Proof sketch of MP in $\mathrm{AxSL}^{\mathrm{SC}}$.

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 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.

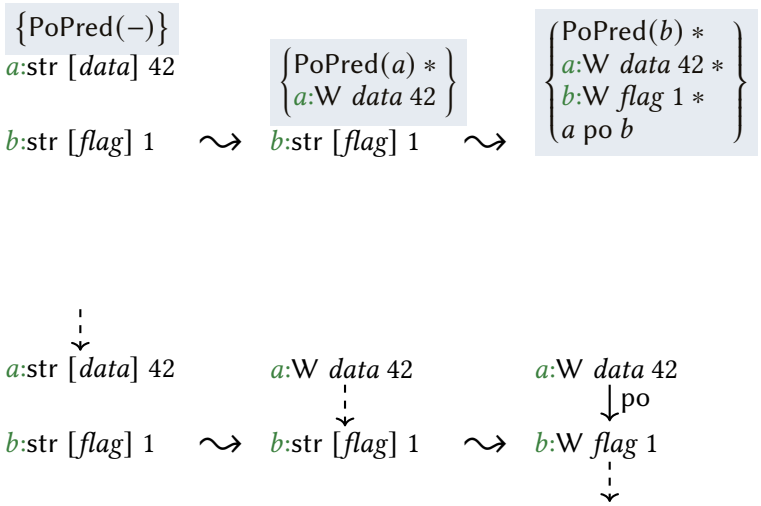


Figure 5.18: The proof (here, of the writer side of MP) in progress (proof steps are indicated by $\rightsquigarrow$) unfolds the program into a graph: either implicitly via assertions in the top row, or visualised as an explicit graph in the bottom row.

We illustrate how to unfold the two instructions to resources, in particular graph facts, in the proof in Figure 5.18.

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.

5.5.3 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 of mixing resources regardless of their sources as in AxSL$^{\mathsf{SC}}$. It is worth noting that this does not add any additional expressiveness power to the logic - in fact AxSL$^{\mathsf{SC}}$ 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$^{\mathsf{SC}}$-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$^{\mathsf{SC}}$ (and other CSLs for SC), we implicitly unify the resource flowing order and the reasoning order, but in AxSL$^{\mathsf{SCExt}}$ we separate them by making the former *explicit* with the tied-to assertion. We elaborate this idea with the AxSL$^{\mathsf{SCExt}}$ version of AxSL$^{\mathsf{SC}}$ rules depicted in Figure 5.13.

Proof Rules for Microinstructions

There are substantial similarities between AxSL$^{\mathsf{SCExt}}$ rules and their AxSL$^{\mathsf{SC}}$ counterparts. We have the same bookkeeping assertions for the immediate po predecessor, which is updated in the postcondition; 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 $e' \hookrightarrow P$, 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 $\ast_{(e'_{\mathsf{po}} \mapsto P_{\mathsf{po}}) \in m} (e'_{\mathsf{po}} \hookrightarrow P_{\mathsf{po}})$, 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 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 ❸, tied to the new memory event e, as ❹. 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.

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

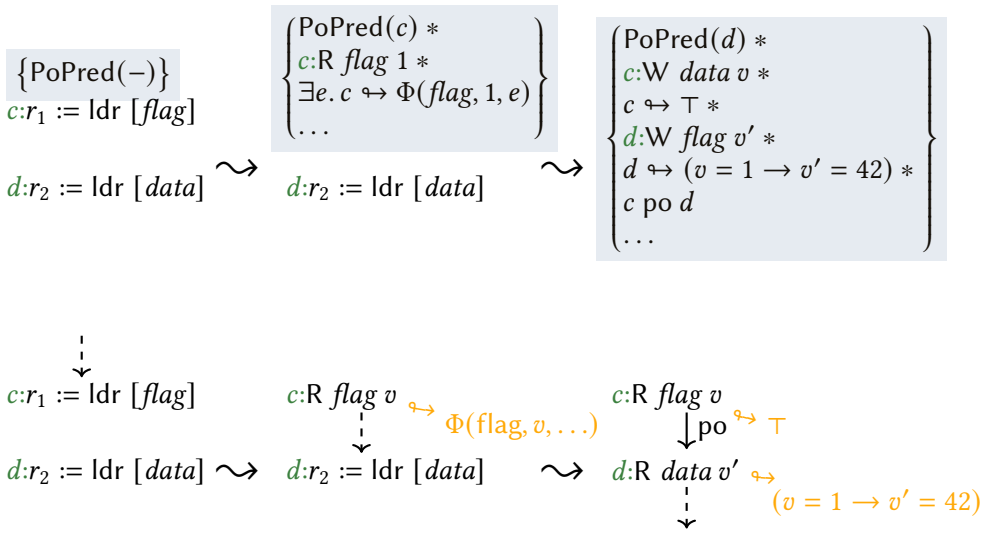


Figure 5.19: The proof (here, of the reader side of MP, in AxSL$^{\text{SCExt}}$) in progress. Resources tied-to a node (in yellow) 'dangle' off it. Between the second $\rightsquigarrow$, we flow $\Phi(\mathit{flag}, v, \ldots)$ from c to d along po, by consuming it from c and using it to conclude $(v = 1 \to v' = 42)$ at d.

in yellow) to account for explicit resource flowing, as follows:

$$r_1 \overset{\mathrm{r}}{\mapsto} v * r_2 \overset{\mathrm{r}}{\mapsto} v' * \exists e.\, e \hookrightarrow (v = 1 \to v' = 42) * (\exists e'.\, e \text{ po } e' \vee e = e' * \text{PoPred}(e'))$$

In general, P holds whenever $a \hookrightarrow P$ appears in a postcondition, since one can flow P from a to a hypothetical terminating event along po. In this new specification, the implication $v = 1 \to v' = 42$ 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 Figure 5.21 (with an illustration in Figure 5.19); the proof of the sending thread is nearly identical to that of AxSL$^{\text{SC}}$ 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 PoPred(c)[3]. Then, we show the flow implication FlowSCR for the load of *data* in line 6 & 7, with P being $v = 1 \to v' = 42$, where we perform the same graph reasoning as in the AxSL$^{\text{SC}}$ MP proof. Finally, we have the desired implication tied to event d, which suffices to show the specification.

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

SCExt-ht-micro-memread

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

$$\mathsf{MemRead}\ x$$

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

SCExt-ht-micro-memwrite

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

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

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

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

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

Figure 5.20: AxSL$^{\mathrm{SCExt}}$ version of the AxSL$^{\mathrm{SC}}$ rules shown in Figure 5.13. The new syntax and changes are highlighted. The user provides m, a thread-local map from events to the resources consumed.

1 $\{\mathsf{PoPred}(-) * r_1 \overset{\mathrm{r}}{\mapsto} \_ * r_2 \overset{\mathrm{r}}{\mapsto} \_\}$

2 $\quad \forall v, e_w. \Phi(\mathit{flag}, v, e_w)$

$\quad \Rrightarrow$

3 $\quad \Phi(\mathit{flag}, v, e_w) * \Phi(\mathit{flag}, v, e_w)$

4 c: r_1 := ldr [flag]

5 $\{\mathsf{PoPred}(c) * \exists v, e. r_1 \overset{\mathrm{r}}{\mapsto} v * c \hookrightarrow \Phi(\mathit{flag}, v, e) * c{:}\mathsf{R}\ \mathit{flag}\ v * e\ \mathsf{rf}\ c\}$

6 $\quad \forall v', e_w. \mathsf{GraphFactsR}(d, \mathit{data}, v', c) * \Phi(\mathit{data}, v', e_w) * \Phi(\mathit{flag}, v, e)$

$\quad \Rrightarrow$

7 $\quad \Phi(\mathit{data}, v', e_w) * (v = 1 \longrightarrow v' = 42)$

8 d: r_2 := ldr [data]

9 $\{\mathsf{PoPred}(d) * \exists v, v'. r_1 \overset{\mathrm{r}}{\mapsto} v * r_2 \overset{\mathrm{r}}{\mapsto} v' * d \hookrightarrow (v = 1 \longrightarrow v' = 42)\}$

Figure 5.21: Proof sketch of the receiving thread of MP in AxSL$^{\mathrm{SCExt}}$. The explicit reasoning of resource flow is highlighted.

5.5.4 Handling Arm-A Concurrency: The AxSL$^{\text{Arm}}$ Logic

Using AxSL$^{\text{SCExt}}$ for TinySc as the base, we build AxSL$^{\text{Arm}}$ for TinyArm. The changes we make to AxSL$^{\text{SCExt}}$ to obtain AxSL$^{\text{Arm}}$ 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$^{\text{Arm}}$, we instead rely on lob, the local fragment of which enforces synchronisation. We elaborate on these two changes with the rule for MemRead in Section 5.5.4.

Proof Rules for Microinstructions

We now explain one of the key proof rules HT-MICRO-MEMREAD-RDEP-EXT in Figure 5.22 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 Figure 5.23.

HT-MICRO-MEMREAD-RDEP-EXT

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

MemRead os vr x d

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

Figure 5.22: A proof rule of AxSL$^{\text{Arm}}$ 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 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

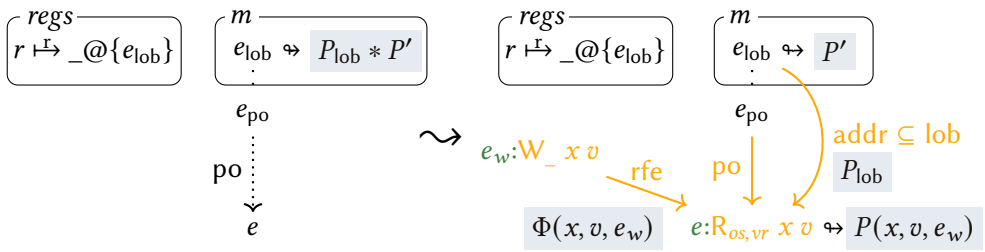


Figure 5.23: A visualisation of HT-MICRO-MEMREAD-RDEP-EXT for external read event e, with logical annotations and newly gained graph facts highlighted. $\rightsquigarrow$ indicates the update of resources from the precondition to the postcondition. The protocol specifies $\Phi(x, v, e_w)$ transferred 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.

resource transfer along obs is possible. This fact is unchanged after the MemRead, and is thus restored by the postcondition as ❽. ❷ $d = (\text{dom}(regs), \emptyset)$ 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 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 CtrlPreds($srcs_{ctrl}$) captures the $srcs_{\text{ctrl}}$ part of the opax thread state T which works like PoPred. The user-supplied m constrained by Po(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 AxSL$^{\text{SCExt}}$ share the same definition.

The postcondition 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.

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 AxSL^{Arm} can be used to reason about Arm-A concurrency compositionally, using two key examples: message-passing (Section 5.5.5) and versions of load buffering (Section 5.5.6), 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 $\text{AxSL}^{\text{SCExt}}$, 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 AxSL^{Arm}. For MP, we only sketch the proof of the receiving thread, highlighting the differences to the MP proof done in $\text{AxSL}^{\text{SCExt}}$. 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.

5.5.5 Graph Reasoning in AxSL^{Arm}: MP+rel+addr

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

Figure 5.24: MP+rel+addr: $r_1 = 1 \longrightarrow r_2 = 42$

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.

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 yellow) at *flag* due to the now relaxed memory model.

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

First, we additionally require in the right disjunct that the write is from Thread 1 (the sending thread) with $\mathit{tid}(e) = 1$. 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 $r_1 \overset{\mathsf{r}}{\mapsto} v@\{c\}$ after reading from the flag, and $c \rightsquigarrow \Phi(\mathit{flag}, v, b)$ 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 $r_2 \overset{\mathsf{r}}{\mapsto} v'@\{d\}$ and $d \rightsquigarrow \Phi(\mathit{data}, v', e)$ for some v' and e, and are required to prove $v = 1 \to v' = 42$. 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.

5.5.6 High-Level Reasoning in AxSL$^{\mathsf{Arm}}$: Load Buffering

We detail how to obtain a high-level proof of LB+artificialdata+data (Figure 5.25), 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 Figure 5.25 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 Section 5.6.5.

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

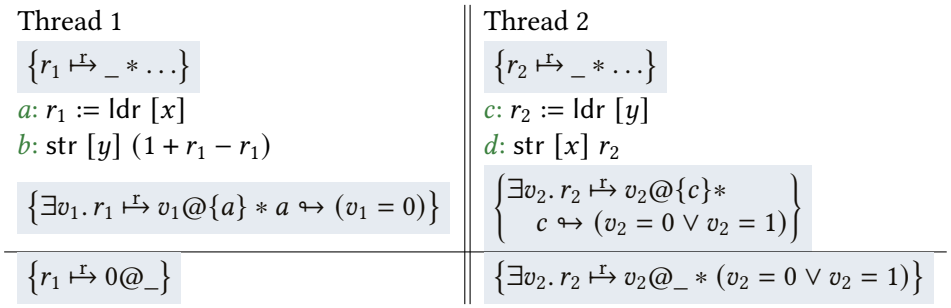


Figure 5.25: LB+artificialdata+data and its (informal) specification

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., 2018b, §2]. The oneshot has two states: pending represents the exclusive permission to make a decision to choose a value, and shot(v) represents the information that the decision has been made to choose value v. In particular, $\boxed{\text{pending}} * \boxed{\text{pending}}$ is a contradiction, and so is $\boxed{\text{pending}} * \boxed{\text{shot}(v)}$, but we can view-shift pending into shot(v), $\boxed{\text{pending}} \Rrightarrow \boxed{\text{shot}(v)}$, to express the logical decision to commit to v. Using this, we can formalise our protocol: for both locations x and y, either the value is 0, or it is 1, in which case we also have $\boxed{\text{shot}(1)}$.

Proof sketch Using this protocol, the proof follows the sketch in Figure 5.26 using the specialised instruction proof rules explained later in Section 5.5.6. 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 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 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

[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 $\Phi(x, v_1, \_)$.

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

Thread 1:

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

2 $(v_1 = 0 \lor (v_1 = 1 * \boxed{\mathsf{shot}(1)})) * \boxed{\mathsf{pending}}$
$\Rrightarrow$
3 $v_1 = 0 * \boxed{\mathsf{pending}}$

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

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

6 $\boxed{\mathsf{pending}}$
$\Rrightarrow$
7 $\boxed{\mathsf{shot}(1)}$

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

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

Thread 2:

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

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

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

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

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

Figure 5.26: Proof sketch of LB+artificialdata+data.

any $\mathsf{shot}(v)$, and thus in particular into $\mathsf{shot}(1)$, as per line 7. This satisfies the flow implication, and thus concludes the proof sketch for Thread 1.

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

Abstraction Using the oneshot resource algebra allows us to reason thread-modularly: the proof of Thread 1 does not involve any graph reasoning about the intricacies of the Arm-A memory model, merely reasoning about abstract state. This is already useful for this small proof sketch (and the corresponding mechanised proof). Thread 2 does not require any inspection of the value or the resource being forwarded, merely plumbing through the flow implication, and the derivation of the contradiction in the impossible case of the load of Thread 1 relies on simple ghost

theory. Moreover, the proof is quite flexible: the proof sketch only requires trivial modifications if (e.g.) we replace the store of $1 + r_1 - r_1$ by a store of r_1.

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 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.

Proof Rules for Instructions

We explain the specialised instruction proof rules used in the LB proof, focusing on (highlighted in yellow) how the rules for Arm-A (Figure 5.27) differ from the (non-specialised) rules for SC (Figure 5.15).

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

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

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

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

HT-INS-STR-PLN-ARTIFICIALDATA

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

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

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

Figure 5.27: Proof rules for the instructions in the left thread of LB+artificialdata+data.

5.5.7 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.

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

$\mathsf{unlock}(\ell) \triangleq \mathsf{str}_{\mathsf{rel}}\ [\ell]$ unlocked

Figure 5.28: Try-lock pseudocode

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

Figure 5.29: 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., 2018b]. 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 Section 5.6.5.

Using our proof rules, we prove the classical specification for simple try-lock (see Figure 5.28), which we then use to prove a basic mutual exclusion example (in Figure 5.29): 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.

5.5.8 Further Examples

To validate how AxSL$^{\mathsf{Arm}}$ 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 (Figure 5.30) relies on the po; [dmb.full]; po clause of bob (see Figure 5.4) to obtain the key lob edge on the left, but the proof is otherwise the same as for LB+artificialdata+data in Section 5.5.6. LB+ctrls (Figure 5.31) relies on the ctrl; [W] clause of dob in both threads, but is otherwise the same. Similarly, MP+rel+dmbsy (Figure 5.32) relies on bob in the right thread, and so does MP+rel+ctrl-isb (Figure 5.33) via the more complex (ctrl|(addr; po)); [ISB]; po; [R] edge which appears incrementally, but their proofs are otherwise as in Section 5.5.5. To illustrate that AxSL$^{\mathsf{Arm}}$ can reason about communication between many threads, we verify an iterated version of MP, namely ISA2+rel+data+acq (Figure 5.34) [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 Section 5.8.4), we verify two coherence tests: CoWW (Figure 5.35) and CoRR (Figure 5.36); the proofs work by symbolic execution followed by discarding the executions with cycles in co.

5.6 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

a: r_1 := ldr $[x]$ || d: r_2 := ldr $[y]$
b: dmb sy || e: str $[x]$ r_2
c: str $[y]$ 1

Figure 5.30: LB+dmbsy+data: $r_1 = 0 \land (r_2 = 0 \lor r_2 = 1)$

a: r_1 := ldr $[x]$ || d: r_2 := ldr $[y]$
b: if $(r_1 == 0)$ || e: if $(r_2 == 1)$
c: str $[y]$ 1 || f: str $[x]$ r_2

Figure 5.31: LB+ctrls: $r_1 = 0 \land (r_2 = 0 \lor r_2 = 1)$

a: str $[data]$ 42 || c: r_1 := ldr $[flag]$
b: str_{rel} $[flag]$ 1 || d: dmb sy
e: r_2 := ldr $[data]$

Figure 5.32: MP+rel+dmbsy: $r_1 = 1 \longrightarrow r_2 = 42$

a: str $[data]$ 42 || c: r_1 := ldr $[flag]$
b: str_{rel} $[flag]$ 1 || d: if $(r_1 == 1)$
e: isb
f: r_2 := ldr $[data]$

Figure 5.33: MP+rel+ctrl-isb: $r_1 = 1 \longrightarrow r_2 = 42$

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

Figure 5.34: ISA2+rel+data+acq: $r_2 = 1 \longrightarrow r_3 = 42$

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 Section 5.7).

Following the structure of the previous section, we build up the model of AxSL^{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^{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 $\text{AxSL}^{\text{SCExt}}$ 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^{Arm}, which shares the same structure as that of $\text{AxSL}^{\text{SCExt}}$, 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

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

Figure 5.35: CoWW: $a \xrightarrow{\text{co}} b$

a: str $[x]$ 42 $\parallel$ b: $r_1 :=$ ldr $[x]$
c: $r_2 :=$ ldr $[x]$

Figure 5.36: CoRR: $r_1 = 42 \longrightarrow r_2 = 42$

AxSL^{Arm} as it is mostly orthogonal to the memory model, thus similar to the one for $\text{AxSL}^{\text{SCExt}}$.

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

5.6.1 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 $v : \text{Val} \subseteq \text{Exp}$
- a notion of physical state $\sigma : \text{St}$ shared among all threads
- a per-thread step relation $\rightarrow_{tid} \subseteq (\text{Exp} \times \text{St}) \times (\text{Exp} \times \text{St})$ 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., 2018b] for a full definition that supports concurrency. Intuitively, $wp\ e\ \{Q\}_{tid}$ requires postcondition Q to hold if e terminates with a value in thread *tid*. A Hoare triple $\{P\}\ e\ \{Q\}_{tid}$ is then defined as $\square(P \mathrel{-\!\!*} wp\ e\ \{Q\}_{tid})$. 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: $\dot{\Rightarrow} Q(e)$
- 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:

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

Formally, it says that for any physical transition $(e, s) \to_{tid} (e', s')$ 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 $\to_{\mathsf{tp}} \subseteq (\mathsf{list}(\mathsf{Exp}) \times \mathsf{St}) \times (\mathsf{list}(\mathsf{Exp}) \times \mathsf{St})$ that specifies the scheduling of threads — all possible interleavings of primitive steps,
- a terminating thread pool trace $([e_0, \cdots, e_n], s_i) \to^*_{\mathsf{tp}} ([e'_0, \cdots, e'_n], s_t)$ 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 postconditions $\varphi_0, \ldots, \varphi_n$, one for each thread

The conclusion is that the postconditions $\varphi_i(e'_i)$ 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 $(e_i, s) \to_{tid} (e'_i, s')$ 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 postcondition by the value case of the weakest precondition.

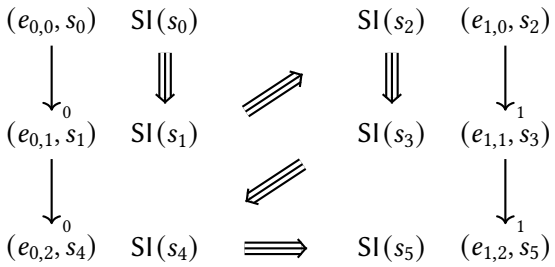


Figure 5.37: An example program execution with two threads, above an SC memory model. $\rightarrow_0$ and $\rightarrow_1$ are the head reductions (the program order and the reasoning order) of the two threads. The indices of s indicate the global order of updating s (induced from $\rightarrow_{\mathsf{tp}}$), 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.

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 $\rightarrow_{\mathsf{tp}}$ 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 local reduction $\rightarrow_{tid}$ of the semantics. Figure 5.37 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: $\mathsf{po} \subseteq \mathsf{sc}$. As discussed in Section 5.3, 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.

5.6.2 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 $\text{AxSL}^{\mathsf{SC}}$, $\text{AxSL}^{\mathsf{SCExt}}$, and $\text{AxSL}^{\mathsf{Arm}}$ respectively.

5.6.3 Model and Soundness of AxSL^{SC}

AxSL^{SC} has a simple semantic model and a one-pass adequacy proof similar to most Iris program logics. AxSL^{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 Figure 5.11 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* $\mathsf{wpb}^{SC}_{tid,\Phi}\ s\ \{Q\}$ that takes an opax state s (the "expression") and a postcondition Q which is a predicate over the terminating state, in the spirit of the aforementioned general recipe, in Figure 5.38.

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

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

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 postcondition of the wp^{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).

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

Figure 5.38: The model of wpb^{SC}. Technical details regarding guarded recursion are omitted.

Overall structure Following the general recipe, the definition of wpb^{SC} has two cases, depending on whether s is a "value". In the value case, we just get the local

state T and assert postcondition $Q(T)$ 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 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 Section 5.5). 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 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 progress by adding e) given it holds on all events in s_{pg} (as ❼), as illustrated in Figure 5.39. 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:

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

The view shift ❼ 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

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

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 ❻ 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.

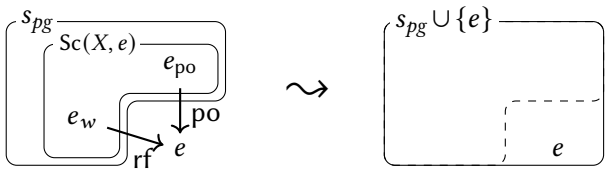


Figure 5.39: An illustration on the evolving of the progress set s_{pg} at a read event e.

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 Figure 5.40.

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

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

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

Figure 5.40: 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., 2023b; 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:

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

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 $\mathsf{AxSL}^{\mathsf{SC}}$:

Theorem 5.6.1. *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 (Figure 5.40) 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 Figure 5.40) and the resource transfer. □

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

The $\mathsf{AxSL}^{\mathsf{SCExt}}$ logic extends the assertion language of $\mathsf{AxSL}^{\mathsf{SC}}$ 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 $\mathsf{AxSL}^{\mathsf{SC}}$. 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 $\text{AxSL}^{\text{SCExt}}$, 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., 2018b]. We present the model in Figure 5.41 and highlight how the model manages tied-to assertions below.

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

Figure 5.41: The model of base weakest precondition of $\text{AxSL}^{\text{SCExt}}$. The key changes to that of AxSL^{SC} are highlighted in yellow. Again, the details handling guarded recursion are omitted.

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', as ❷) and the associated tied assertions. Crucially, the update has to follow the flow implication ❸ 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.

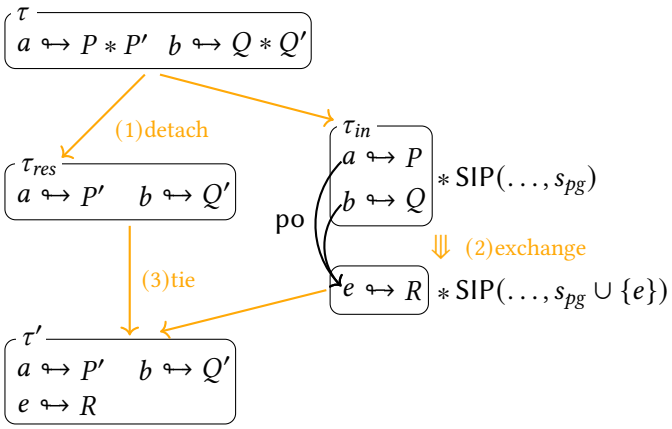


Figure 5.42: A visualisation of the three actions of updating τ to τ', where we flow P and Q to e along po from a and b respectively. R is eventually tied to e after exchange.

We divide the update from resource map τ to τ' into a sequence of three actions: detach, exchange, and tie, as illustrated in Figure 5.42. The ❹ 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 $\mathrm{AxSL}^{\mathrm{SC}}$. It now has the local resources of τ_{in} explicitly given on the left as ❻, and the resource ❼ R remaining on event e on the right. The map update ❺ does the tying: the remaining resource R is *tied* to e, by extending the map τ_{res}.

Stratification

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

[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.

The effect of tying P to a in AxSL$^{\mathsf{SCExt}}$ is to constrain the use of P: it isolates P from the reasoning along po that we do using wp$^{\mathsf{SCExt}}$. This stratification is reflected in the model wpb$^{\mathsf{SCExt}}$: the update of tied resources as performed by the view shift in FlowImp cannot affect the recursive wpb$^{\mathsf{SCExt}}$ for s': we cannot use the updated resource to directly reason about s'. In Figure 5.41, this is reflected by wpb$^{\mathsf{SCExt}}$ being pulled out of the view shift. This is to contrast with AxSL$^{\mathsf{SC}}$ (see Figure 5.38), where 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 AxSL$^{\mathsf{Arm}}$, as we will show in the next section.

Soundness of Proof Rules

Like in 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 5.6.2. *The AxSL$^{\mathsf{SCExt}}$ proof rules for microinstructions are sound.*

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

The model for AxSL$^{\mathsf{SCExt}}$ is almost parametric in the memory model, in the sense that it almost works directly for AxSL$^{\mathsf{Arm}}$, a logic for TinyArm with relaxed Arm-A concurrency. We elaborate the two key changes we make to adapt the 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.

Hoare Triples

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

The model of the base weakest precondition wpb$^{\mathsf{Arm}}$ has the same overall structure as in 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 wpb$^{\mathsf{Arm}}$, as depicted in Figure 5.43:

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

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

Figure 5.43: The model of base weakest precondition of AxSL$^{\mathsf{Arm}}$. The diff from AxSL$^{\mathsf{SCExt}}$ (highlighted in yellow) reflects the shift of the synchronisation order from sc to ob.

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 5.6.3. *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 Section 5.6.4), and we need to deal with the dependency edges of Arm-A. □

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 Figure 5.8). Modelling such splitting is, however, quite challenging due to its higher-order nature. 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., 2018b]. 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^{Arm} protocols, like escrows in GPS [Kaiser et al., 2017; Turon et al., 2014].

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 Section 5.5.

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 $srcs_{\mathsf{rmw}}$ (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 $srcs_{\mathsf{ctrl}}$ with 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 uniquness of a token

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

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 $\mathrm{wpb}^{\mathsf{Arm}}$: we replace the $\dot{\Rrightarrow} Q$ in the case handling termination with $\mathsf{PullOutTied}(\mathit{tid}, Q)$:

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

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 Figure 5.25.

Stuckness and Infinite Executions

As described in Section 5.4.3, 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 Section 5.4.3), our definitions of weakest preconditions does not take measures to handle infinite executions either.

5.7 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/Rocq). 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 Section 5.6.1. The adequacy proofs of $\mathrm{AxSL}^{\mathsf{Arm}}$ differs from — and, we argue, generalises — the general recipe (basically, the general recipe is our approach, instantiated with the same order twice, see Section 5.7.2 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^{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^{Arm}; we skip $\text{AxSL}^{\text{SCExt}}$, since it has an almost identical semantic model, and thus proof, to AxSL^{Arm}. This proof significantly differs from the previous general recipe, due to the new semantic model that enforces stratification (as described in Section 5.6.4). 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.

5.7.1 Adequacy of AxSL^{SC}

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

Theorem 5.7.1 (Adequacy of AxSL^{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*

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

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 $\mathsf{Ctd}\ \vec{C}$. The second line assumes that we have proofs in AxSL^{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 postconditions are assumed to be the meta-level propositions $\vec{P}$, lifted to AxSL^{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^{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.

The value of the adequacy theorem is that it means that we do not need to trust or even understand the intricacies of AxSL^{SC}: once we have proved weakest preconditions for each thread using the AxSL^{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).

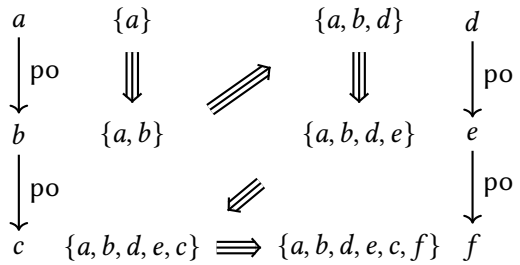


Figure 5.44: An example program execution graph with two threads, above an SC memory model. Assuming sc edges from b to d, from e to c, and from c to f, $\Rrightarrow$ indicates the order we update $\mathsf{SIP}(\Phi, X, s_{pg})$, the interpretation of the progress set in the adequacy proof, where the sets of nodes indicates the advance of the progress omitting initial nodes.

Proof Sketch. The overall proof strategy is that we first show $\dot{\models}\left\lceil \bigwedge_{tid=1}^{n} \vec{P}(tid) \right\rceil$, that is the goal lifted to AxSL^{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 Figure 5.44. 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 $s_{pg} \supseteq \mathsf{Pred}(X.\text{sc}, e)$ 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.

5.7.2 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 5.7.1, 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 any events. However, the *proof* of adequacy significantly diverges from that of Theorem 5.7.1, because of the stratification enforced by the semantic model of AxSL$^{\mathsf{Arm}}$.

Theorem 5.7.2 (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*

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

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

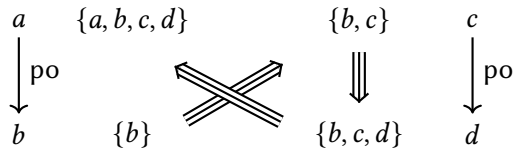


Figure 5.45: An example program execution graph with two threads, above Arm-A memory model. Assuming ob edges from b to c, from c to d, and from d to a, $\Rrightarrow$ indicates the order we update $\mathsf{SIP}(\Phi, X, s_{pg})$ in the second phase of the adequacy proof, where the sets of nodes indicate the advance of s_{pg}.

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 $\cup$ 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 $\cup$ 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^{Arm}. However, we still want syntax-oriented weakest preconditions which collect flow implications along po, as illustrated in Figure 5.45. This tension poses the main challenge to the proof of adequacy, since one cannot do induction on the potentially cyclic po $\cup$ 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 AxSL^{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 (Section 5.9).

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 5.7.3. *For a valid execution graph X,*

$$\begin{aligned}
&\exists \varepsilon, \tau. \textcircled{1}\mathsf{SI}_{\mathsf{T}}(\tau) * (\textcircled{2}\mathsf{dom}(\tau) = \mathsf{dom}(\varepsilon) = \mathsf{AllNodes}(X)) * \\
&\quad (\textcircled{3}\forall e \mapsto m \in \varepsilon.\, \mathsf{dom}(m) \subseteq \mathsf{PredOf}(X.\mathsf{lob}, \mathsf{e})) * \\
&\quad \forall e \mapsto R \in \tau. \\
&\qquad \forall s_{pg} \supseteq \mathsf{PredOf}(X.\mathsf{ob}, e) \wedge e \notin s_{pg}. \\
&\qquad\quad \left(\mathsf{SIP}(\Phi, X, s_{pg}) * \textcircled{4} \circledast_{_ \mapsto R_i \in \varepsilon[e]} R_i\right) \Rrightarrow \\
&\qquad\qquad \left(\mathsf{SIP}(\Phi, X, s_{pg} \cup \{e\}) * \textcircled{5} \circledast_{e_o \in \mathsf{SuccOf}(X.\mathsf{lob}, \mathsf{e})} \varepsilon[e_o][e] * R\right)
\end{aligned}$$

Generally speaking, the lemma requires a well-formed annotation on every lob edge of the execution graph. The edge annotation ε (of type $Eid \to Eid \to iProp$) records the history of how resources evolve and are transferred soundly along lob in a complete program execution (e.g. $\varepsilon[e][e']$ 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 $\varepsilon[e]$ 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. □

Phase Two

Given the edge annotation ε from phase one, phase two stitches the flow implications together by induction on ob, as illustrated in Figure 5.45.

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. □

Adequacy of AxSL$^{\text{SCExt}}$

The adequacy proof of AxSL$^{\text{SCExt}}$ differs from that of AxSL$^{\text{Arm}}$ 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$^{\text{SC}}$, but divided into two phases due to the stratification imposed by the model. Recall that in the AxSL$^{\text{SC}}$ 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.

5.8 Technical Remarks and Limitations

5.8.1 Technical Improvements to the Original AxSL$^{\text{Arm}}$

Although the syntax of AxSL$^{\text{Arm}}$ in Section 5.5 is almost identical to the original AxSL$^{\text{Arm}}$ [Hammond et al., 2024], the model presented in Section 5.6 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$^{\text{SCExt}}$.

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) 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.

5.8.2 Recovering Points-Tos in AxSL^{SC}

In this subsection, 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^{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 Section 5.8.2. Our goal is to obtain a definition of points-to that satisfies the same rules in AxSL^{SC}.

STD-HT-MICRO-MEMREAD
$\{x \hookrightarrow v\}\ \mathsf{MemRead}\ x\ \{w.\ w = v * x \hookrightarrow v\}_{tid,\Phi}$

STD-HT-MICRO-MEMWRITE
$\{x \hookrightarrow v\}\ \mathsf{MemWrite}\ x\ v'\ \{\_.\ x \hookrightarrow v'\}_{tid,\Phi}$

Figure 5.46: 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.

Raw Definition

In AxSL^{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 $\mathsf{WriteOf}(x, e)$ 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.

SC-PT-RAW-AG
$$\frac{x \hookrightarrow_{raw} v@e * \mathsf{WriteOf}(x, e')}{e'\ \mathsf{co}^*\ e}$$

SC-PT-RAW-SHT
$$x \hookrightarrow_{raw} v@e \Rrightarrow \mathsf{WriteOf}(x, e)$$

SC-PT-RAW-UPD
$$\frac{e\ \mathsf{co}\ e' * e'{:}\mathsf{W}\ x\ v'}{x \hookrightarrow_{raw} v@e \Rrightarrow x \hookrightarrow_{raw} v'@e'}$$

SC-PT-WO-PERS
$$\frac{\mathsf{WriteOf}(x, e)}{\mathsf{WriteOf}(x, e) * \mathsf{WriteOf}(x, e)}$$

Figure 5.47: Selected operations of the raw points-to and WriteOf

Full Definition

The full points-to assertion is modelled as a monotone predicate over the po-latest event e of a thread, using the raw definition:

$$[\![x \hookrightarrow v]\!] \triangleq \lambda e.\, \exists e_{lst}.\, x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e$$

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$^{\mathsf{SC}}$ logic. Most of them are standard, for instance below is how we monotonise the weakest precondition.

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

Proving Standard CSL Rules

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

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

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

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):

$$\begin{array}{l}\forall e_w.\, (\mathsf{GraphFactsW}(e_w, x, v', e) * \exists e_{lst}.\, x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e) \\ \qquad \Rrightarrow\ (\mathsf{WriteOf}(x, e_w)) * \exists e_{lst}.\, x \hookrightarrow_{raw} v'@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e_w\end{array}$$

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:

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

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:

$$\begin{array}{l}\forall e_r, w, e_w.\, \left(\begin{array}{l}\mathsf{GraphFactsR}(e_r, x, v, e_w, e) * \\ \exists e_{lst}.\, x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e * \mathsf{WriteOf}(x, e_w)\end{array}\right) \\ \qquad \Rrightarrow\ w = v * \exists e_{lst}.\, x \hookrightarrow_{raw} v@e_{lst} * e_{lst} \leq_{\mathsf{sc}} e_r * \mathsf{WriteOf}(x, e_w)\end{array}$$

By SC-PT-RAW-AG, we know $e_w \leq_{\mathsf{co}} e_{lst}$, 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.

5.8.3 Proof Effort for AxSL$^{\mathsf{Arm}}$AxSLArm

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 Section 5.8.1, 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/Rocq.

Overall, the mechanisation for AxSL$^{\text{Arm}}$ 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.

5.8.4 Coherence

The memory model of Arm-A involves two main axioms: `external`, which requires ordered-before to be acyclic, and `internal`, which requires `po-loc | ca | rf` to be acyclic, which effectively enforces per-location sequential consistency (the atomicity axiom is much more 'local' and easier to use, as per Section 5.5.7). 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$^{\text{Arm}}$ is defined in Iris above the opax semantics using the memory model of Figure 5.4 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$^{\text{Arm}}$ 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$^{\text{Arm}}$, by explicit graph reasoning (as in Section 5.5.5), but this is unsatisfactory. This is a definite limitation of AxSL$^{\text{Arm}}$, as many common programming and reasoning idioms rely on reasoning about coherence, in particular full GPS protocols and the notion of non-atomics.

5.9 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 Figure 5.4). 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 Section 5.5.1, 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 Section 5.8.4. 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]. Program $(\text{declare-const } x); s$ 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., 2015b] 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.

5.10 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.

6

First Steps towards AxSL+

6.1 Introduction

Very relaxed memory models, such as the one employed in Arm-A, underpin many high-performance systems and mobile devices. They challenge the conventional assumptions that the union of program order (po) and the reads-from relation (rf) is acyclic, therefore require advanced methodologies for program logic verification.

AxSL [Hammond et al., 2024; Liu et al., 2024] is an Iris-based concurrent separation logic framework that can be instantiated with such very relaxed memory models and support local reasoning and higher-order ghost states. A key feature of AxSL is that it enables sound resource transfer along the synchronisation order that does not respect program order. In particular, AxSL^{Arm}, its instance for Arm-A, is proven sound with respect to the synchronisation order of Arm-A, which many program logics built for stronger memory models assuming $\text{po} \cup \text{rf}$ acyclicity cannot handle. Despite the advancements, reasoning about synchronisation order (ob) and coherence (eco) of Arm-A *simultaneously* remains an open problem [Hammond et al., 2024; Liu et al., 2024], which is necessary to offer expressive abstractions capturing programming patterns of realistic concurrent code.

This work introduces AxSL+, an enriched AxSL^{Arm} aiming for tackling this problem. For brevity, we omit the superscript indicating the memory model from the logic names, referring to AxSL^{Arm} as AxSL throughout this paper. By extending and generalising AxSL with a novel *level-indexing* mechanism, we present AxSL+^{NA}, a simple instance of AxSL+ that supports non-atomic points-to assertions that are transferable along both ob and eco. This marks the first instance of *mixed-order thread-local reasoning*, opening the door for more expressive abstractions requiring passing resources along multiple orders. The level-indexing mechanism is also generic and applicable to other multi-order memory models.

The structure of the paper is as follows:

- Section 6.2 discusses the extended coherence axiom of Arm-A and highlights the primary challenge of reasoning about it alongside ob via a double message-passing (Double MP) example.

```
1  (* SC-per-location *)
2  acyclic po-loc | ca | rf
3  (* External visibility *)
4  irreflexive ob
```

Figure 6.1: The two "axioms" of Arm-A axiomatic model by Deacon [Deacon, 2016; Pulte et al., 2018], where `ca` is defined as `fr | co`

- Section 6.3 provides a technical overview of the level-indexing solution to this challenge.
- Section 6.4 introduces AxSL+$^{\mathsf{NA}}$, a simple AxSL+ instance with non-atomic points-to assertions, including its assertion language, specialised proof rules, and a proof sketch of Double MP.
- Section 6.5 presents the semantic models for the weakest precondition and the implementation of AxSL+$^{\mathsf{NA}}$.
- Section 6.6 outlines the adequacy proof of AxSL+.
- Section 6.7 discusses potential future improvements to AxSL+$^{\mathsf{NA}}$.

Remarks We provide additional technical details in this paper to enhance the credibility of our results, as they are not yet mechanised. We assume that readers are familiar with the AxSL papers.

6.2 Context

This section first recalls the extended coherence axiom of Arm-A, then explains why mixing eco reasoning with ob reasoning to AxSL is necessary but hard.

6.2.1 Coherence in Arm-A model

The Arm-A memory model employs the ordered-before (ob) order as its synchronisation order. It also enforces coherence, the standard condition that concurrent memory accesses to each location exhibit sequentially consistent behaviour, commonly referred to as SC-per-location. This coherence condition is formally expressed as the acyclicity of the union of four orders: po-loc$\cup$rf$\cup$co$\cup$fr, as shown in Figure 6.1. For convenience, we refer to the union of these orders as eco (extended-coherence), distinguishing it from the standard coherence order co, which only concerns writes.

While the definition of eco in Arm-A is standard, the fact that it is not included in ob means that reasoning rules for eco-based abstractions do not arise as easily as they do in stronger models in which eco is included in ob. Specifically, such logics easily allow flowing resources *between* ob and eco. However, this is unsound in Arm-A because such flowing relies on acyclicity of ob $\cup$ eco, as we recall below in Section 6.2.3.

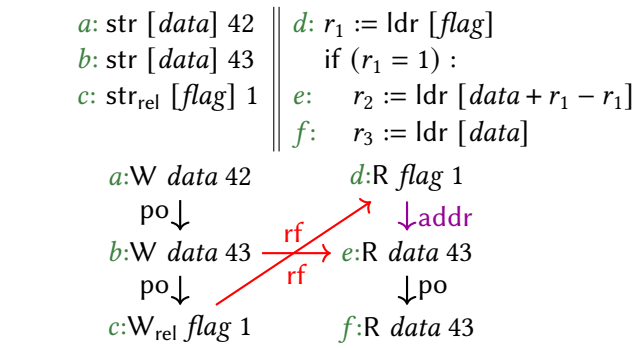


Figure 6.2: DMP+rel+addr: $r_1 = 1 \Rightarrow r_2 = r_3 = 43$

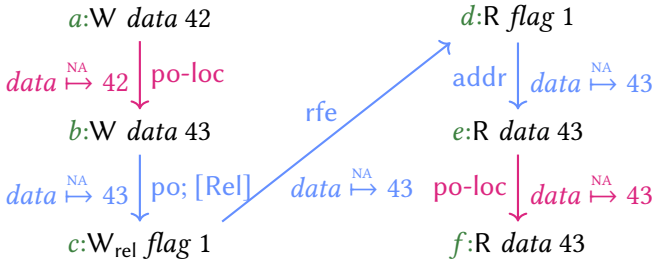


Figure 6.3: The path that the non-atomic assertion for *data* flows and updates along

6.2.2 Mixing ob and eco Is Necessary

To understand why allowing resource flow along ob ∪ eco is necessary, consider verifying the Double MP example in Figure 6.2 in a hypothetical Arm logic with eco-based abstractions. In this example, the *data* location is free of data races, enabling the use of the so-called 'non-atomic points-to' to track its value.

Non-atomic points-tos

Non-atomic locations (free of data races) can be reasoned about using sequential consistency with non-atomic points-to assertions. $data \overset{\text{NA}}{\mapsto} v$ ensures only v can be read from *data*, prohibiting reads of outdated values to maintain coherence. Reasoning rules for non-atomic locations require exclusive ownership of the assertion for loads and stores, which prevents data races on them.

Focusing on one execution where the read d of the flag reads 1 from c (execution on the right in Figure 6.2), we identify a path from the first write to the data, a, to the last read of the data, g, that the non-atomic points-to assertion for *data* would flow and update along as in Figure 6.3. In this path, the points-to assertion first updates at the writes a and b to the data, passing between the two events along po-loc. Next, to transfer the assertion to the reader thread, it flows to the write c to the flag along po; [*Rel*]. At the write c to the flag, a resource transfer mechanism such as a per-location protocol sends the points-to to the read d of the flag along

rfe. After the read d of the flag receives the assertion, it flows to the first read f of the data for reading along addr, and finally to the second read g of the data along po-loc. This path involves both ob and eco, specifically:a eco b ob c ob d ob f eco g. Generally speaking, such switching pattern between ob and eco can occur multiple times when using eco-based abstractions like non-atomic points-tos, and is the key to make abstractions of this kind usable in the logic.

6.2.3 Mixing ob and eco Is Unsound

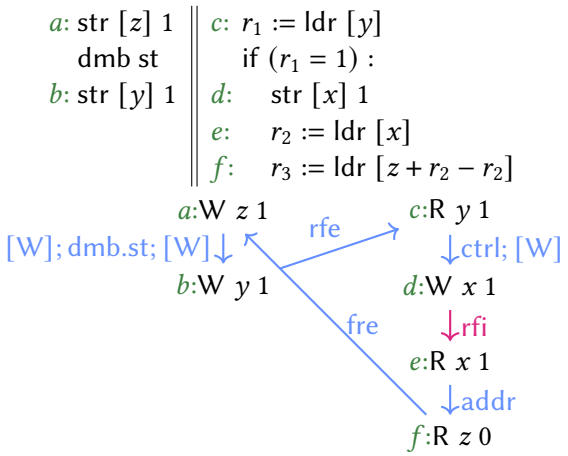


Figure 6.4: PPOCA: $r_1 = 1 \wedge r_3 = 0$ is allowed

Adding eco-based abstractions introduces a broader circularity issue arising from mixing eco and ob. The absence of an acyclicity condition for eco$\cup$ob allows circular dependencies in Arm-A. The *allowed* execution of PPOCA in Figure 6.4 manifests an eco $\cup$ ob cycle: a ob b ob c ob d eco e ob f ob a. Allowing to flow resources along such a cycle is clearly unsound.

6.2.4 Solution: Stratification

Resolving the circularity issue requires a flexible stratification strategy beyond AxSL's solution that only permit unidirectional resource exchange from po to ob. Our proposed solution employs a generic stratification strategy that permits bounded bi-directional resource exchange between eco and ob. We believe this approach subsumes AxSL's more restrictive solution and is broadly applicable to scenarios involving multiple orders.

6.3 Technical Overview

The major technical contributions of AxSL+ is a new semantic model that allows mixed reasoning of both ob and eco using stratification. In this section, we provide an intuitive explanation of the key ideas underpinning AxSL+.

We first recall how AxSL achieves sound ob reasoning by relying on flow implications, detailed in Section 6.3.1. We then demonstrate how the same principle can be adapted to eco, facilitating isolated eco reasoning as an initial step towards mixed-order reasoning, as discussed in Section 6.3.2. Once we have introduced this context, in Section 6.3.3, we delve into the unsoundness that arises from naïvely combining reasoning for the two orders. We then outline our solution using stratification: we index resources with *levels* which count how many switches between orders have been performed. Finally, we discuss the formalisation of our solution and how it manifests in proof rules in Section 6.3.4.

6.3.1 AxSL Recap: Sound ob Reasoning with Flow Implications

The fundamental principle to ensure soundness of a relaxed memory logic with ghost resources is to only allow passing and updating resources along some order, typically the synchronisation order. This approach is central to AxSL for Arm-A, where the synchronisation order is ob. To allow syntax-directed reasoning along program order, AxSL employs some indirection using two key ideas: *flow implications* inspired by RSL, and its novel *tied-to assertions*.

A tied-to assertion $e \hookrightarrow P$ serves as a fine-grained modality guarding P, which can only be used by events that are ob-later than e. A flow implication for an event e is essentially a resource update (technically, an Iris view-shift) guarded by a *flow condition* C_{ob}:

$$C_{\mathsf{ob}}(e, P) \twoheadrightarrow (P \Rrightarrow Q * R)$$

Here, C_{ob} ensures that the view-shift is applicable only when the incoming resource P comes from events ob-ordered before e. Q and R denote the resources obtained after the update, where R remains tied to e and Q flows out to other events.

We illustrate the flow implication as an annotated execution graph fragment around e as in Figure 6.5, where ob edges are annotated by the resources flowing along them respectively.

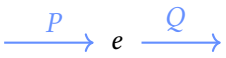


Figure 6.5: Flow implication for the event e where e is abstracted as the node, with hypothetical ob edges pointing to and from the node. P is the resource flowing in to e; Q is flowing out after the update $P \Rrightarrow Q * R$; R is the resource staying on e, which is omitted.

The process of reasoning about a program in AxSL involves collecting the flow implications for all events within the execution graph of the program. The soundness of the logic is validated by the existence of a fully annotated execution graph, constructed by stitching together annotated fragments, each representing a flow implication for an event. Formally, this entails creating a chain of resource updates by using the flow implications. This construction is at the core of AxSL's two-step adequacy proof, which first uses induction along program order to collect flow implications, and then follows this chain construction by induction on the synchronisation order, ob. This ensures that the C_{ob} condition holds at each induction step.

6.3.2 Adding Support for Just eco

The initial step towards AxSL+ involves adding *independent* support for eco to AxSL, effectively constructing a logic solely for eco reasoning, and combining it with AxSL for ob as a disjoint union. This simplification allows us to initially avoid addressing the interaction between the two orders, which we address in Section 6.3.3.

A key observation is that the sound reasoning along ob provided by flow implications is independent of the specific definition of the order: it depends solely on the acyclicity of ob, which offers an induction principle. The 'internal' coherence (SC-per-location) axiom of Arm-A ensures such an acyclicity condition for eco, and so we can use the same approach along eco. To do this, we merely instantiate the order-parameterised flow condition C with eco, denoted as C_{eco}. This results in a new flow implication for eco, expressed as:

$$C_{\mathsf{eco}}(e, P) \mathrel{-\!\!*} (P \Rrightarrow Q * R)$$

This flow implication enables tracking resource flows along eco.

By incorporating this flow implication into AxSL alongside the one for ob, we establish the semantic foundation for isolated eco support. On the logic side, we enrich AxSL's assertion language, particularly the tied-to assertion, which tracks resources tied to specific events. To accommodate both orders, we refine the syntax by parameterising the tied-to assertion with an order $\mathcal{R} \in \{\mathsf{ob}, \mathsf{eco}\}$. This gives us two types of tied-to assertions, $e_{\mathsf{ob}} \rightsquigarrow P$ and $e_{\mathsf{eco}} \rightsquigarrow P$, which track resources tied to the two orders *separately*. Intuitively, $e_{\mathcal{R}} \rightsquigarrow P$ means that P is tied to event e after a sequence of flows and updates along $\mathcal{R}$, and it can flow to events that are $\mathcal{R}$-after e.

While these semantic and syntactic extensions enable independent reasoning for eco and ob, it does not yet provide a mechanism for transferring resources *between* the two orders. However, such an exchange mechanism is essential, as argued in Section 6.2.2, to achieve mixed-order reasoning.

6.3.3 Handling The Mix of the Two Orders

As we have pointed out in Section 6.2.3, mixing resources from two orders is generally unsound because their union might be cyclic, which means that we cannot perform

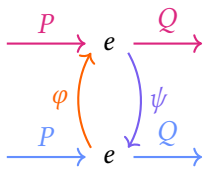


Figure 6.6: Two node-internal edges for flowing between the two orders

induction on it. Therefore, to exchange resource between the two orders, some restrictions are required to break the potentially circularity. Before we delve into our solution, we show more concretely the circularity issue we would encounter if we approached the problem naïvely.

Circularity with explicit cross-order edges

The issue becomes evident when we explicitly represent the desired resource exchange between eco and ob at any node. Figure 6.6 illustrates this exchange using two special internal edges (internal as the two nodes represent the same physical event in the graph): resources flowing from ob to eco are denoted by $\rightarrow$ annotated with φ, and the reverse direction by $\rightarrow$ annotated with ψ. We refer to these internal edges as *order-switching edges*.

Clearly, adding these order-switching edges naïvely creates cycles in the graph, leading to unsound resource flow paths. The two order-switching edges can create cycles by themselves, enabling resources to flow from a node back to itself. Order-switching edges can also connect acyclic ob and eco paths to form cycles, as $\mathsf{ob} \cup \mathsf{eco}$ is not guaranteed to be acyclic in a consistent execution graph (as shown in Section 6.2.3). Thus, resource flows between the two orders must be constrained to ensure soundness.

Breaking circularity with level-indexing

In AxSL, the graph was annotated with a layer of resources. In AxSL+, to allow sound mixing of reasoning along different orders, we use two ideas.

Levels First, we introduce some stratification via a notion of integer *levels*. Instead of one layer of annotations, we have multiple layers, effectively adding a new dimension of annotations on top of the execution graph, as in Figure 6.7 (where we repeat the graph unchanged at each level for ease of reference). Each annotation (solid blue edges for ob annotations, and solid magenta edges for eco annotations) is located at a given level, which indicates how many order-switching have been taken to establish it.

Order-switching edges Second, we introduce order switching edges (solid orange edges for ob to eco, and purple edges for eco to ob), which go from a given node at

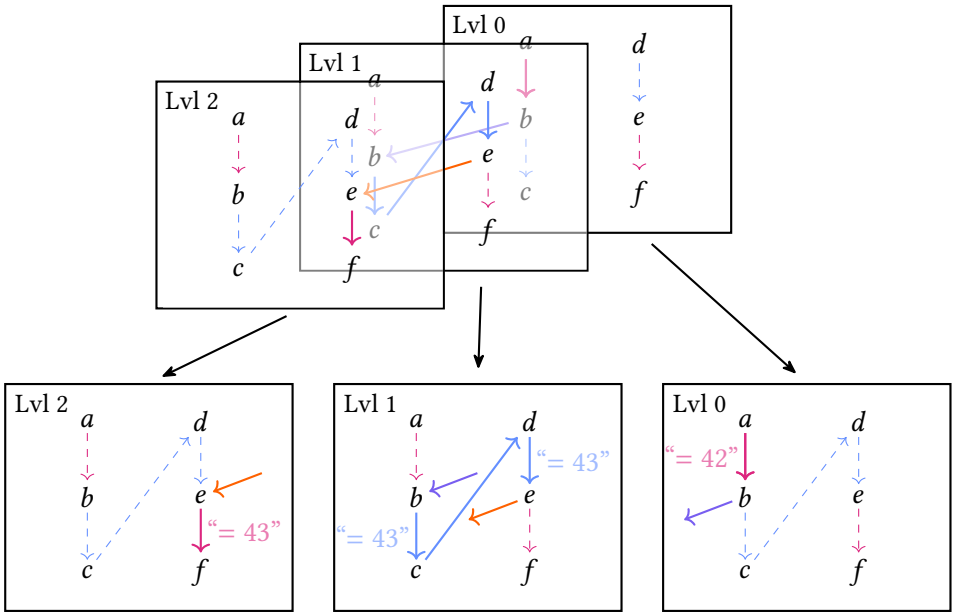


Figure 6.7: Illustration of the path a eco b ob c ob d ob e eco f through Double MP of Figure 6.3 with level-indexing.
We show two views: with all levels together with the complete path through them at the top, and with each level individually with its partial path (including incoming and outgoing partial edges) at the bottom.
Because there are two order switchings, we have three levels. We repeat the graph at each level for ease of reference, highlighting with solid edges the annotations at that level (the dashed edges are merely shown for convenience). Annotations are represented in solid blue edges for ob and solid magenta edges for eco. Order switching (at b from level 0 to level 1, and at e from level 1 to level 2) is represented in solid orange edges for ob to eco, and purple edges for eco to ob.

a lower level to itself at a higher level, along this new dimension.

Together, this ensures that, when flowing along a path, going through an order-switching edge strictly increases the level of the resources, thus ruling out cycles.

Implementing level indexing

In AxSL+, we implement this idea of level indexing by changing flow implications to be level-indexed. For node e at given level k, we collect the following two symmetric flow implications:

$$\begin{array}{rlllll} & k & k-1 & & k & k+1 \\ C_{\text{ob}}(e,k,P_{\text{ob}}) \twoheadrightarrow & P_{\text{ob}} * & \varphi & \Rrightarrow & Q_{\text{ob}} * & \psi' \\ C_{\text{eco}}(e,k,P_{\text{eco}}) \twoheadrightarrow & P_{\text{eco}} * & \psi & \Rrightarrow & Q_{\text{eco}} * & \varphi' \end{array}$$

The first implication states that ob resource P_{ob} at level k can combine with eco resource φ from level $k-1$ to produce two resources: Q_{ob} in ob at level k, and ψ' in eco at level $k+1$. Figure 6.8 shows this flow diagrammatically, with two planes for ob and eco.

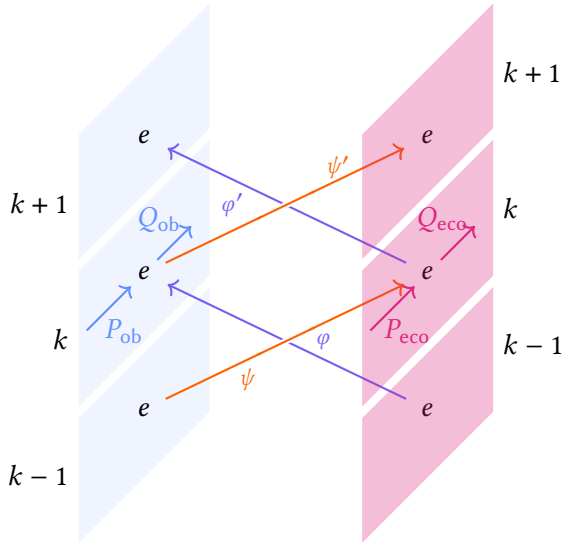


Figure 6.8: Level-indexing resources flowing through event e at level k.
Layers of annotations $k-1$, k, and $k+1$ are represented horizontally. We separate the ob flow on the left from the eco flow on the right, with order-switching edges going across (orange edges for ob to eco, and purple edges for the opposite direction).

There are two caveats to level level-indexing. First, to draw a conclusion in AxSL+, we need to put an upper bound $\mathcal{K}$ to the order switching, on which AxSL+ is parameterised. Although there are cycles in ob $\cup$ eco, AxSL+ only considers finite paths through them. Users of the logic need to pick an appropriate upper bound for each individual example they want to verify with the logic.

Second, we can add a logical state to each order and leverage the fact that flow implications have to be applied in a sequence respecting the corresponding order to make sure that the involvement of the state also respects the order. For instance, we can maintain and update a list of history of writes for every location along eco. This is the key to implement eco-abstractions, which we return to in Section 6.5.

6.3.4 Formalising Level-indexing Rules

In this section, we discuss how we formalise level-indexing in AxSL+. We introduce level-indexing through novel *level-and-order-indexed tied-to assertions* and *level-indexed flow implications* that work together by following a set pattern.

Level-and-order-indexed tied-to assertions Our new tied-to assertion $e \overset{k}{\underset{\mathcal{R}}{\hookrightarrow}} P$ is indexed not only by an event e, but also by a level k, and by an order $\mathcal{R}$. Intuitively,

this new assertion means that P is tied to e after k order-switchings, and can flow to other events at the same level k along order $\mathcal{R}$, or events at higher levels after order-switching.

We now demonstrate how the monotonicity of level-indexing is implemented by restricting the update of the new tied-to assertions, via a base pattern that all proof rules of the AxSL+ follow. Concrete proof rules incorporating mixed-order reasoning are derived by specialising this base pattern, as explained in Section 6.4.

In precondition:

$$\left(\textcircled{1}\circledast_{((k,e)\mapsto P)\in m_{\text{lob}}} (e^k_{\text{ob}} \rightsquigarrow P)\right) * \left(\textcircled{2}\circledast_{((k,e)\mapsto P)\in m_{\text{po-loc}}} (e^k_{\text{eco}} \rightsquigarrow P)\right) *$$
$$\forall e, \cdots. \left(\begin{array}{l}\mathsf{GraphFacts}(e, \cdots) \mathrel{-\!\!*} \\ (\textcircled{3}\mathsf{Lob}(\mathsf{dom}(m_{\text{lob}}), e, \mathcal{K}) * \textcircled{4}\mathsf{PoLoc}(\mathsf{dom}(m_{\text{po-loc}}), e, \mathcal{K}) * \\ \quad \textcircled{7}\mathsf{FlowImp}(e, m_{\text{lob}}, m_{\text{po-loc}}, t_{\text{ob}}, t_{\text{eco}}, \mathcal{K}, 0, \top))\end{array}\right)$$

In postcondition:

$$\exists e.\, \mathsf{GraphFacts}(e, \cdots) *$$
$$\left(\textcircled{5}\circledast_{(k\mapsto P)\in t_{\text{ob}}} e^k_{\text{ob}} \rightsquigarrow P(\cdots)\right) * \left(\textcircled{6}\circledast_{(k\mapsto P)\in t_{\text{eco}}} e^k_{\text{eco}} \rightsquigarrow P(\cdots)\right)$$

Figure 6.9: A sketch of the general pattern for manipulating the new tied-to assertions in proof rules.

Pattern Figure 6.9 shows the base pattern of the update of tied-to assertions specified by the pre- and postcondition of a proof rule for some event e. Generally, this pattern extends that of AxSL by adding levels to all the relevant resource-flowing assertions for ob, and then duplicating this for eco. The pattern manages ob resource-flowing in the same manner as AxSL, but now works with level-indexed tied-to assertions for ob: clauses ❶, ❸, and ❺ are as in AxSL, but level-indexed. Clause ❶ now requires all ob tied-to assertions specified by m_{lob}, a thread local map of type $Lvl \to Eid \to iProp$, such that $m_{\text{lob}}[k][e']$ is the ob resource currently tied to event e' at level k, and will flow to the current event along lob, a local ob edge. Predicate Lob of ❸ is strengthened to also check whether all the levels in the mappings of m_{lob} are valid, namely smaller than the upper bound $\mathcal{K}$ (in addition to the existing lob predecessor check of AxSL). We will see in the next section that the Hoare triples used to formulate proof rules are also parameterised by the same upper bound $\mathcal{K}$. Similarly, t_{ob}, which specifies the final resources tied to the current node e in the postcondition is now also parameterised by levels, as per clause ❺. These three components together effectively tracks how resources tied to ob predecessors (m_{lob}) flow along ob edges and attach to e as t_{ob} after certain updates (and order switchings with eco).

Symmetrically, we now have ❷, ❹, ❻ handling eco reasoning. The key component connecting the resources of the two orders is the new flow implication ❼, which

$$\mathsf{FlowImp}(e, m_{\text{lob}}, m_{\text{po-loc}}, t_{\text{ob}}, t_{\text{eco}}, \mathcal{K}, k, R) \triangleq$$

$$\begin{array}{l}(\mathcal{K} < k \to R) * \\ \left(\begin{array}{l}\mathcal{K} \geq k \to \\ \textcircled{1}\exists U_{\text{ob}}, U_{\text{eco}}. \\ \textcircled{2}\left(\left(\ast_{(\_\mapsto P_{in}) \in m_{\text{lob}}[k]} P_{in}\right) \Rrightarrow \ (t_{\text{ob}}[k] * U_{\text{ob}})\right) * \\ \textcircled{3}\left(\left(\ast_{(\_\mapsto P_{in}) \in m_{\text{po-loc}}[k]} P_{in}\right) \Rrightarrow \ (t_{\text{eco}}[k] * U_{\text{eco}})\right) * \\ \textcircled{4}\left((U_{\text{ob}} * U_{\text{eco}}) \mathrel{-\!\!*} \mathsf{FlowImp}(e, m_{\text{lob}}, m_{\text{po-loc}}, t_{\text{ob}}, t_{\text{eco}}, \mathcal{K}, k+1, R)\right)\end{array}\right)\end{array}$$

Figure 6.10: Simplified level-indexed Flow Implication of AxSL+

regulates the resource exchange between ob and eco by enforcing the monotonicity principle. Informally, it requires one to show that the transformation of the tied-to assertions represented by the following transformation of the maps is valid up to level $\mathcal{K}$:

$$\langle m_{\text{lob}}, m_{\text{po-loc}} \rangle \rightsquigarrow \langle t_{\text{ob}}, t_{\text{eco}} \rangle$$

Flow implication Figure 6.10 depicts a simplified but formal version of how predicate FlowImp implements the idea of flowing to higher levels when switching orders, and Figure 6.11 shows a diagrammatic representation. The full version used in the semantic model also allows logical state updates, and is presented in Section 6.5.

At its core, FlowImp is a recursive definition that roughly repeats the original flow implications of AxSL for the two orders, from k to $\mathcal{K}$. We start from the lowest level k and increment it until reaching $\mathcal{K}$. In the terminating case, we require ownership of resource R, which works as a continuation, in the sense that R is the accumulated resource coming from lower levels, which can be passed to levels higher than $\mathcal{K}$. This generic design enables flexibility that is especially induction friendly. In the case where k is not greater than $\mathcal{K}$, we perform a local check at k which consists of three actions:

- First, we *specify* U_{ob} and U_{eco}, which are the resources that will flow to higher levels (❶) from both orders.
- We then (on lines ❷ and ❸) *check* the validity of the tied-to map update

  $$\langle m_{\text{lob}}, m_{\text{po-loc}} \rangle \rightsquigarrow \langle t_{\text{ob}}, t_{\text{eco}} \rangle$$

  only at level k using flow implications for the two orders separately. The iterated separation conjunctions on the left of the view shifts are the resources flowing in. Resources $t_{\text{ob}}[k]$ and $t_{\text{eco}}[k]$ on the right are the resources tied to event e afterwards. They further ensure that U_{ob} and U_{eco} are indeed available after the resource update, such that it can be consumed later at higher levels. These two lines are roughly of the form described in Section 6.3.3, with two subtleties. First, resources φ and ψ flowing from lower levels, which were

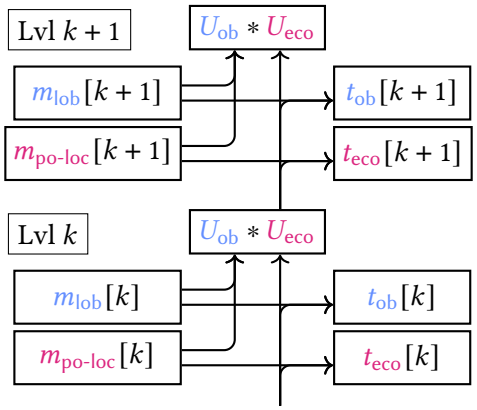


Figure 6.11: An illustration of the simplified FlowImp at level k and $k+1$, edges connecting the blocks are the permitted resource flowing paths.

explicitly shown on the left of the view shifts in Section 6.3.3, are now implicitly assumed thanks to this recursive definition. Second, U_{ob} and U_{eco} on the right of the view shifts include resources that may flow to *any* event at higher levels, including but not limited to φ' and ψ' that flow to the node itself.

- Finally, we *provide* U_{ob} and U_{eco} to higher levels by placing them on the left of the separating implication ❹, with the recursive occurrence of FlowImp for $k+1$ on the right.

6.4 An AxSL+ Instance for Non-Atomics: AxSL+^{NA}

In this section, we build AxSL+^{NA}, a simple instance of AxSL+ specialised to non-atomic reasoning, to validate the level-indexing idea presented in Section 6.3. In particular, we use it to verify the Double MP example while avoiding the explicit graph reasoning that AxSL had to resort to.

AxSL+^{NA} is merely one possible instance of AxSL+, whose level-indexing mechanism enables a whole category of logics that can do mixed-order reasoning. In fact, the semantic model that we develop in Section 6.5 and use for this instance is generic and flexible, and we expect it can be used as the base of more elaborate logics.

The AxSL+^{NA} logic presented in this section relies on that model to support eco-based non-atomics for succinct local reasoning and ob-based protocols for resource transfer. AxSL+^{NA} builds upon the same Arm-like assembly language and its opax semantics from AxSL, but introduces significant enhancements through extensions to the assertion language and the addition of new proof rules.

In Section 6.4.1, we present the extensions made to the assertion language. Then, we elaborate on eco-based non-atomic points-tos and ob-based protocols, demonstrating their integration into the tied-to update pattern of Section 6.3.4 to

$$
\begin{aligned}
P, Q \in \mathit{iProp} ::= {} & e^{k}_{\mathcal{R}} \hookrightarrow P && \text{Tied-to} \\
\mid {} & \{P\}\, i\, \{Q\}_{\mathit{tid},\mathcal{K}} && \text{Hoare triple} \\
\mid {} & x \overset{\mathsf{NA}}{\mapsto} v_{[e]} && \text{Non-atomic points-to} \\
\mid {} & x :\!\!: \mathcal{R} && \text{Order typing} \\
\mid {} & \mathrm{Prot}(x, \Phi) && \text{Protocol} \\
\mid {} & r \overset{\mathsf{r}}{\mapsto} v@E \mid \text{(other AxSL connectives)} \cdots
\end{aligned}
$$

$$
v \in \mathit{Val} \qquad e \in \mathit{Eid} \qquad x \in \mathit{Addr} \qquad r \in \mathit{Reg}
$$

$$
k, \mathcal{K} \in \mathit{Lvl} \qquad \mathcal{R} \in \{\mathsf{ob}, \mathsf{eco}\}
$$

$$
\Phi \in \mathit{Val} \to \mathit{Lvl} \to \mathit{Eid} \to \mathit{iProp}
$$

Figure 6.12: The assertion language of AxSL+, the extension to the language of AxSL is highlighted. $e^{k}_{\mathcal{R}} \hookrightarrow P$ is pronounced as P is tied to event e at level k for order $\mathcal{R}$. $x \overset{\mathsf{NA}}{\mapsto} v_{[e]}$ is pronounced as x non-atomic points to v witnessed by event e. $x :\!\!: \mathcal{R}$ is pronounced as x is of order type $\mathcal{R}$. $\mathrm{Prot}(x, \Phi)$ is pronounced as x is governed by protocol Φ.

create streamlined high-level proof rules in Section 6.4.2. Finally, we showcase the rules by proving the Double MP example.

6.4.1 Assertion Language

The assertion language of AxSL+ extends AxSL through small but powerful generalisations of existing assertions, and the introduction of new predicates. Figure 6.12 presents the assertions of AxSL+.

The extension of the assertion language consists of two components: first is parameterising the tied-to assertion and Hoare triples with levels to enable bound resource transfer between orders, as discussed in Section 6.4; second is the introduction of new predicates to facilitate eco and ob reasoning. We elaborate their semantics when we encounter them in the proof rules in Section 6.4.2.

6.4.2 Specialised Proof Rules

Like AxSL, AxSL+$^{\mathsf{NA}}$ uses two layers of proof rules: a base layer of generic but complex rules, and a surface layer of specialised but more user-friendly rules. For simplicity, we present specialised proof rules for the specific instructions used in the Double MP example of Figure 6.2 that we will use in Section 6.4.3.

In AxSL+$^{\mathsf{NA}}$, we have two kinds of proof rules: rules for non-racy locations, and rules for racy locations. For the first kind, we support eco via non-atomic points-tos. For the second kind, we support resource transfer along ob using protocols. We rely on the order typing assertion $x :\!\!: \mathcal{R}$ to indicate which kind of rules is applicable

for location x. This is a persistent assertion, meaning that this decision has to be made upfront, and all threads have to agree with it. It is in principle also possible to generalise AxSL+$^{\mathsf{NA}}$ to reason about coherence for racy locations, or to reason about two orders for a location; we remark on these in Section 6.7.

Proof rules are formulated with the new level-indexed Hoare triple $\{P\}\ i\ \{Q\}_{tid,\mathcal{K}}$, which intuitively means that instruction i of thread tid updates resource P to Q, given the upper bound $\mathcal{K}$ for order-switching. The protocol used to index Hoare triples in AxSL is turned into a logical assertion.

eco-based non-atomics

Figure 6.13 depicts two rules for specific relaxed load and store that demonstrate the use of non-atomic points-to assertions. The non-atomic points-to $x \overset{\mathsf{NA}}{\mapsto} v_{[e]}$

HT-STR-RLX-NA-ECO

$$\left\{ \begin{array}{l} \mathsf{PoPred}(e_{\mathsf{po}}) * x \overset{\circ}{:} \mathsf{eco} * \textcircled{1} e_0 \overset{k}{\underset{\mathsf{eco}}{\rightsquigarrow}} (x \overset{\mathsf{NA}}{\mapsto} v_{0[e_0]}) * \textcircled{2}(k \le \mathcal{K}) * \\ \forall e, v.\ \big(\mathsf{GraphFacts}(e, x, v, e_{\mathsf{po}}) \mathrel{-\!\!*} \textcircled{3}(e_0\ \text{po-loc}\ e)\big) \end{array} \right\}$$

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

$$\left\{ \exists e.\ \mathsf{PoPred}(e) * \mathsf{GraphFacts}(e, x, e_{\mathsf{po}}) * \textcircled{4} e \overset{k}{\underset{\mathsf{eco}}{\rightsquigarrow}} \left(x \overset{\mathsf{NA}}{\mapsto} v_{[e]}\right) \right\}_{tid,\mathcal{K}}$$

HT-LDR-RLX-NA-OB-ADDR

$$\left\{ \begin{array}{l} \mathsf{PoPred}(e_{\mathsf{po}}) * x \overset{\circ}{:} \mathsf{eco} * \textcircled{5} e' \overset{k}{\underset{\mathsf{ob}}{\rightsquigarrow}} (x \overset{\mathsf{NA}}{\mapsto} v_{[e_0]}) * \textcircled{6}(k < \mathcal{K}) * \\ r \overset{\mathsf{r}}{\mapsto} \_@\_ * r' \overset{\mathsf{r}}{\mapsto} \_@\{e_0\} \end{array} \right\}$$

$$r := \mathsf{ldr}\ [x + r' - r']$$

$$\left\{ \begin{array}{l} \exists e, v_0.\ \mathsf{PoPred}(e) * \mathsf{GraphFacts}(e, x, v, e_{\mathsf{po}}) * \\ \textcircled{7} e \overset{k+1}{\underset{\mathsf{eco}}{\rightsquigarrow}} (x \overset{\mathsf{NA}}{\mapsto} v_{[e]} * \textcircled{8} v_0 = v) * r \overset{\mathsf{r}}{\mapsto} v_0@\{e\} * r' \overset{\mathsf{r}}{\mapsto} \_@\{e_0\} \end{array} \right\}_{tid,\mathcal{K}}$$

Figure 6.13: AxSL+ proof rules for relaxed accesses, highly specialised for use a on non-atomic locations.

intuitively means that the value of x is v, and the latest write has been witnessed by a local event e in a manner that prevents reading from outdated writes.. An implementation of this assertion can be found in Section 6.5.2. Like regular points-tos, holding this assertion means holding the exclusive permission to read and write to this location, thus ruling out races.

We now take a closer look at HT-STR-RLX-NA-ECO, which updates the value of the points-to to v for location x with a store. To apply this rule, one has to provide the points-to assertion $x \overset{\mathsf{NA}}{\mapsto} v_{0[e_0]}$ that is currently tied to some node e_0 at level k, as in ❶. After the store, one would get $x \overset{\mathsf{NA}}{\mapsto} v_{[e]}$ tied to the current node e at the same level k, as in ❹. We do not change the level as there is no order-switching happening. To apply this rules, there are two side conditions: first, the level k is in the scope

that the triple considers (❷); and second, e_0 and e are ordered by po-loc (❸), which allows the resource flow.

HT-LDR-RLX-NA-OB-ADDR is a specialised rule for a load that has an artificial address dependency on register r'. We have to bump the level here since an order-switching from ob to eco is happening. The non-atomics points-to assertion flow from e_0 in ob at level k (❺) to e in eco at level $k+1$ (❼) along an $\mathsf{addr} \in \mathsf{lob}$ edge. Importantly, the equality between the result value of register r and the value of the points-to v (❽) is inside the tied-to assertion, meaning one cannot conclude it now, but only at level $k+1$. This is impractical, but is required for soundness: one of the goals of tied-to assertions is to enforce stratification, which is fundamental to tackle the circularity problem. Like before, there is a side condition for levels: ❻ ensures that the bumped value is still in bounds.

ob-based protocols

HT-LDR-RLX-AT

$$\left\{\begin{array}{l}\mathsf{NoLocalWrites}(y) * \mathsf{PoPred}(e_{\mathrm{po}}) * y \,\colon\!\!\circ\, \mathsf{ob} * \mathsf{Prot}(y,\Phi) * \\ ❶(\forall k \in \mathsf{dom}(t).\, k \le \mathcal{K}) * \\ \forall e, v, e_w.\, \left(\begin{array}{l}\mathsf{GraphFacts}(e,y,v,e_w,e_{\mathrm{po}}) \mathrel{-\!\!*} \\ *_{(k\mapsto Q)\in t}\ (❷\Phi(v,k,e_w) \Rrightarrow \Phi(v,k,e_w) * Q(v,k,e_w))\end{array}\right)\end{array}\right\}$$

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

$$\left\{\begin{array}{l}\exists e, v, e_w.\, \mathsf{NoLocalWrites}(y) * \mathsf{PoPred}(e) * \mathsf{GraphFacts}(e,y,v,e_w,e_{\mathrm{po}}) * \\ \left(❸ *_{(k\mapsto Q)\in t}\ e^k_{\mathsf{ob}} \hookrightarrow Q(v,k,e_w)\right)\end{array}\right\}_{tid,\mathcal{K}}$$

HT-STR-REL-AT

$$\left\{\begin{array}{l}\mathsf{NoLocalWrites}(y) * \mathsf{PoPred}(e_{\mathrm{po}}) * y \,\colon\!\!\circ\, \mathsf{ob} * \mathsf{Prot}(y,\Phi) * \\ \left(❹ *_{((k,e)\mapsto P)\in m} (e^k_{\mathsf{eco}} \hookrightarrow P)\right) * (\forall (k,\_) \in \mathsf{dom}(m).\, k \le \mathcal{K}) \\ \forall e, v.\, \left(\begin{array}{l}\mathsf{GraphFacts}(e,y,v,e_{\mathrm{po}}) \mathrel{-\!\!*} \\ \left(❺ *^{\mathcal{K}}_{k=0} \left((*_{(\_\mapsto P)\in m[k-1]}\ P) \Rrightarrow \Phi(v,k,e)\right)\right)\end{array}\right)\end{array}\right\}$$

$$\mathsf{str}_{\mathsf{rel}}\ [y]\ v$$

$$\left\{\ \exists e.\, \mathsf{LastLocalWrite}(y,e) * \mathsf{PoPred}(e) * \mathsf{GraphFacts}(e,y,v,e_{\mathrm{po}})\ \right\}_{tid,\mathcal{K}}$$

Figure 6.14: AxSL+ proof rules for relaxed reads and writes, highly specialised for protocols.

For racy location y, we enforce ob protocol Φ on it with assertion $\mathsf{Prot}(y,\Phi)$. Unlike in AxSL, where it is indexed by only a value and an event ID, Φ here is additionally indexed by a level, making it possible to transfer different resources at different levels. The way resources are transferred is similar to that of AxSL, except that now it does so for multiple levels.

HT-LDR-RLX-AT is a load rule that receives resources specified by a map t of type $Lvl \to (Val \to Lvl \to Eid \to iProp)$ from an external write e_w (indicated by NoLocalWrites). For every mapping $k \mapsto Q$ of t, one has to show that $Q(v, k, e_w)$ can be derived given the protocol resources $\Phi(v, k, e_w)$ sent by e_w as shown in ❷. The received resources are tied-to event e at their levels in the postcondition as ❸, as long as the levels of t are in bounds (❶).

HT-STR-REL-AT is a rule for release write that transfers eco resources specified by a finite map m (❹, of type $Lvl \to Eid \to iProp$) to other ob events reading from this write. The tied-to resources of m flow to current write e along lob (this is given because this write is a release write). We require that the ob protocol at all levels are satisfied, so that when receiving the resources one has the freedom to choose the (one or more) levels at which one receive them. Since an order-switching is happening, when showing required protocol resources at level k, one can only use eco resources of m at level $k - 1$, as seen in ❺. In general, it does not have to be merely $k - 1$, but can be anything lower than k; we show this specialised version for simplicity.

$$\Phi : Val \to Lvl \to Eid \to iProp$$

$$\Phi(1_V, 1_L, e) \triangleq \boxed{\left(\forall e_r.\, e \ \mathsf{rfe}\ e_r \twoheadrightarrow data \overset{\text{NA}}{\mapsto} 43_{[e_r]}\right) \vee \diamond}$$

$$\Phi(1_V, \_, \_) \triangleq \mathsf{Emp}$$

$$\Phi(0_V, \_, \_) \triangleq \mathsf{Emp}$$

$$\Phi(\_, \_, \_) \triangleq \bot$$

Figure 6.15: Protocol for Double MP. We sometimes use subscripts L and V for the type of constants Lvl and Val to avoid ambiguity.

6.4.3 Double MP

With the proof rules presented above in hand, we now verify the Double MP example of Figure 6.2 in a *thread-local* manner. We show a proof sketch in Figure 6.16, and a diagram in Figure 6.17. In this example, we know that there are no races on location *data*, and so we use non-atomic points-to for it. We use ob reasoning for location *flag*, and set up its protocol as in Figure 6.15. The protocol transfers, at level 1, the non-atomic assertion of *data* with value 43 wrapped in an invariant, when the *flag* is set. This invariant implements a simple escrow pattern: one can obtain the points-to in exchange for an exclusive token $\diamond$. We let the reader thread hold the token so that it can take the points-to out of the invariant and use it for the two reads of data.

We set the upper bound as 2 as there are two order-switchings happening in the proof. Figure 6.16 contains a sketch of the proof.

Thread 1:

1 $\left\{e_{0\,\mathsf{eco}}^{0} \rightsquigarrow data \overset{\mathsf{NA}}{\mapsto} 0_{[e_0]} * \mathsf{IsInit}(data, e_0) * \ldots\right\}$

2 *a*: str $[data]$ 42

3 $\left\{a_{\mathsf{eco}}^{0} \rightsquigarrow data \overset{\mathsf{NA}}{\mapsto} 42_{[a]} * \ldots\right\}$

4 *b*: str $[data]$ 43

5 $\left\{b_{\mathsf{eco}}^{0} \rightsquigarrow data \overset{\mathsf{NA}}{\mapsto} 43_{[b]} * \ldots\right\}$

6 $data \overset{\mathsf{NA}}{\mapsto} 43_{[b]} * b \text{ lob } c$
$\Rrightarrow$

7 $\boxed{\left(\forall e_r.\, c \text{ rfe } e_r \mathrel{-\!\!*} data \overset{\mathsf{NA}}{\mapsto} 43_{[e_r]}\right) \vee \diamond}$

8 *c*: $\mathsf{str}_{\mathsf{rel}}$ $[flag]$ 1

9 $\{\ldots\}$

Thread 2:

10 $\{\diamond * r_1 \overset{\mathsf{r}}{\mapsto} \_ * r_2 \overset{\mathsf{r}}{\mapsto} \_ * r_3 \overset{\mathsf{r}}{\mapsto} \_ * \mathsf{NoLocalWrites}(flag) * \ldots\}$

11 $\boxed{\left(\forall e_r.\, c \text{ rfe } e_r \mathrel{-\!\!*} data \overset{\mathsf{NA}}{\mapsto} 43_{[e_r]}\right) \vee \diamond} * \diamond * c \text{ rfe } d$
$\Rrightarrow$

12 $data \overset{\mathsf{NA}}{\mapsto} 43_{[d]}$

13 *d*: $r_1 := \mathsf{ldr}\ [flag]$

14 $\left\{\exists v.\, r_1 \overset{\mathsf{r}}{\mapsto} v@\{d\} * d_{\mathsf{ob}}^{1} \rightsquigarrow \left(v = 0_V \vee v = 1_V * data \overset{\mathsf{NA}}{\mapsto} 43_{[d]}\right) * \ldots\right\}$

15 if $r_1 = 1$:

16 $\left\{r_1 \overset{\mathsf{r}}{\mapsto} 1@\{d\} * d_{\mathsf{ob}}^{1} \rightsquigarrow \left(data \overset{\mathsf{NA}}{\mapsto} 43_{[d]}\right) * \ldots\right\}$

17 *f*: $r_2 := \mathsf{ldr}\ [data + r_1 - r_1]$

18 $\left\{\exists v_2.\, r_2 \overset{\mathsf{r}}{\mapsto} v_2@\{f\} * f_{\mathsf{eco}}^{2} \rightsquigarrow \left(v_2 = 43 * data \overset{\mathsf{NA}}{\mapsto} 43_{[f]}\right) * \ldots\right\}$

19 *g*: $r_3 := \mathsf{ldr}\ [data]$

20 $\left\{\begin{array}{l}\exists v_2, v_3.\, r_3 \overset{\mathsf{r}}{\mapsto} v_3@\{g\} * g_{\mathsf{eco}}^{2} \rightsquigarrow \left(v_3 = 43 * data \overset{\mathsf{NA}}{\mapsto} 43_{[g]}\right) * \\ r_2 \overset{\mathsf{r}}{\mapsto} v_2@\{f\} * f_{\mathsf{eco}}^{2} \rightsquigarrow (v_2 = 43) * \ldots\end{array}\right\}$

Figure 6.16: A thread-local sketch proof of Double MP. The lines whose numbers are in green are the worth-noting side proof obligations. We omit the predetermined and global knowledge $x :\!\!: \mathsf{eco} * y :\!\!: \mathsf{ob} * \mathsf{Prot}(y, \Phi)$, and the upper bound of 2. We skip e as the event label to avoid confusion with variable e used in binders. We do not show the last step of pulling out tied resources.

In thread 1, we start with the points-to for *data* at level 0 with value 0 from the initial write e_0. By applying HT-STR-RLX-NA-ECO, we update the value of *data* to 42 at write a. We apply the same rule again for the second write b. After that, on line 5, we have the points-to with value 43 tied to b at level 0. Finally, use HT-STR-REL-AT for the release write on *flag*, picking the map $m = \{(0, b) \mapsto \mathit{data} \overset{\text{NA}}{\mapsto} 43_{[b]}\}$. We have to show that we can satisfy the ob protocol Φ of *flag* at level 0 to 2. Level 0 and 2 are trivial, line 6 and 7 shows the level 1 case where we show the protocol $\Phi(1_V, 1_L, c)$ by allocating the invariant using the points-to assertion. When establishing the invariant, we show that we can update the witness of the points-to assertion from b to any external read e_r of c, which is instantiated by d in the other thread. Here we can use the points-to assertion at level 0 because there is a order-switching from eco to ob.

In thread 2, we start with the exclusive token $\diamond$, and the assumption that there is no local write to *flag*, which allows ob resources transfer along rfe. We apply HT-LDR-RLX-AT for the read on line 13, picking $t = \{1_L \mapsto (\lambda v\ k\ e.\ (v = 0_V) \lor (v = 1_V * \mathit{data} \overset{\text{NA}}{\mapsto} 43_{[d]}))\}$. This gives us the proof obligation $(v = 0_V) \lor (v = 1_V * \mathit{data} \overset{\text{NA}}{\mapsto} 43_{[d]}) * \Phi(v, 1_L, c)$ given protocol resource $\Phi(v, 1_L, c)$ at level 1. The left disjunct of case $v = 0_V$ is trivial; line 11 and 12 justify the right disjunct when $v = 1_V$. In the latter case, we open the invariant and exchange the exclusive token with the points-to assertion. Next, the branching allows us to eliminate the left disjunct, such that the points-to is tied to d at level 1 at line 16. We proceed with the HT-LDR-RLX-NA-OB-ADDR rule for the read of *data* with address dependency from d, which allows us to flow the points-to assertion from level 1 to f at level 2 due to order-switching. At line 18, we obtain the register assertion for r_2 with value v_2 and equality $v_2 = 43$ tied to f at level 2. We conclude the proof with a variant of the non-atomic load rule that flows resources along eco without order-switching, which allows us to show that r_3's value is $v_3 = 43$ tied-to g at level 2.

Figure 6.17 depicts the proof visually, focusing on the flowing path of the non-atomic points-to we have seen in Figure 6.3, but lifted into the level-indexing setup.

6.5 Semantic Model

This section introduces the semantic model of AxSL+, built using the Iris base logic. We use this model to establish the soundness of AxSL+. We demonstrate the soundness of all proof rules with respect to this model and prove its adequacy theorem, ensuring reliable verification results by excluding AxSL+ from the trusted base. We touch on the first aspect at the end of this section, and defer the latter to Section 6.6.

We first present the base weakest precondition model in Section 6.5.1, emphasising general aspects that support sound mixed-order reasoning while keeping interpretation predicates abstract for simplicity. In Section 6.5.2, we instantiate these abstract predicates, define new assertions, and mention the key properties they

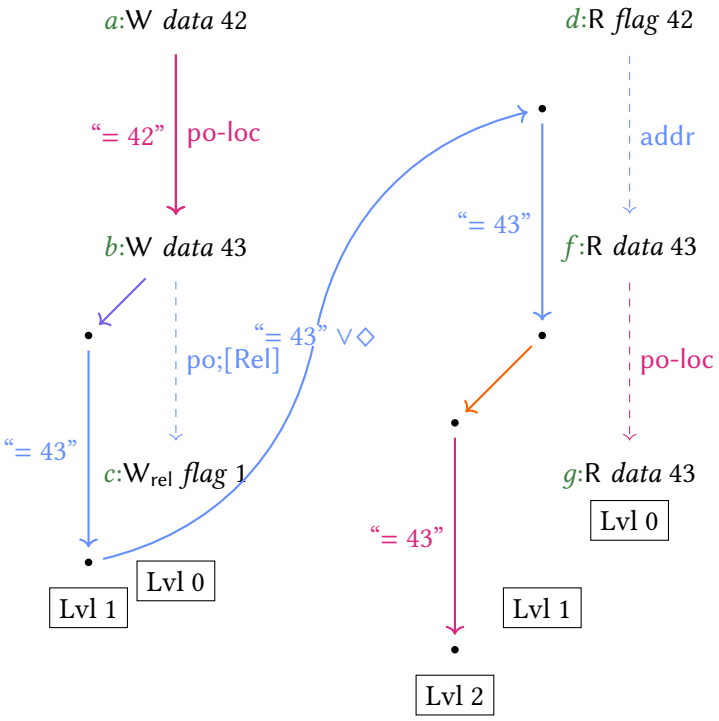


Figure 6.17: Illustration of the flow of resources in the proof of Double MP of Figure 6.3 with level-indexing. We start the proof at level 0 at event *a*, flow the resource tracking the value of the *data* location along the depicted path, and stop at level 2 at event *g*. The same-thread paths are established during the thread-local proofs for the two thread, and the different-thread path from event *c* to *d* is established by the protocol for *flag*.

satisfy, which are essential to the soundness of the proof rules. Finally, Section 6.5.3 states the soundness theorem.

6.5.1 Semantic Model of Weakest Preconditions

This subsection details the logical foundation of the *base* weakest precondition in AxSL+. The base weakest precondition operates on a local machine state *s* which can often be abstracted away through appropriate local constructs, as in the Hoare triples used to formulate the proof rules. We direct readers to the original AxSL paper for the abstractions implemented by other weakest preconditions and Hoare triples, as their definitions remain unchanged from AxSL.

The base weakest precondition in Figure 6.18 retains the structure of AxSL, with notable distinctions highlighted in yellow. These include the handling of level-indexed tied-to assertions with the novel level-indexed flow implication FlowImp. We elaborate on these components below while referencing AxSL for the unchanged part. It is worth noting that, in the definitions, the treatments of ob and eco are

$$\mathsf{wpb}^{\mathcal{K}}_{tid}\ s\ \{Q\} \triangleq$$

$$\begin{array}{l}
\big(s = \mathsf{Done}\ T \wedge \Mapsto \mathsf{PullOutTied}(tid, Q(T))\big) \vee \\
\left(\begin{array}{l}
s = \mathsf{Ctd}\ C \wedge \\
\left(\begin{array}{l}
\forall X, I.\, \mathsf{Valid}(X) \mathrel{-\!\!*} (\Box\, \mathsf{GI}(\langle X, I\rangle)) \mathrel{-\!\!*} \forall s'.\, \langle s, \langle X, I\rangle\rangle \xrightarrow{tid}_{\mathsf{h}} \langle s', \langle X, I\rangle\rangle \mathrel{-\!\!*} \\
\quad \textcircled{11}\forall O.\, (\Box\, \mathsf{SI}_{\mathsf{O}}(O)) \mathrel{-\!\!*} \forall e = \langle tid, C.T.IT.cntr\rangle. \\
\mathit{Case}\ \mathsf{ValidEvt}(e, X):\ \\
\quad \forall \textcircled{1} T_{\mathsf{ob}}, T_{\mathsf{eco}}.\, \textcircled{2}\mathsf{TI}_{\mathsf{ob}}(T_{\mathsf{ob}}) * \textcircled{3}\mathsf{TI}_{\mathsf{eco}}(T_{\mathsf{eco}}) \Mapsto \exists \textcircled{4} T'_{\mathsf{ob}}, T'_{\mathsf{eco}}.\, \mathsf{TI}_{\mathsf{ob}}(T'_{\mathsf{ob}}) * \mathsf{TI}_{\mathsf{eco}}(T'_{\mathsf{eco}}) * \\
\quad\quad \textcircled{5}\exists S_{\mathsf{ob}}, S_{\mathsf{eco}}.\, (\mathbin{*}_{\mathcal{R}}\, \mathsf{WFIf}_{\mathcal{R}}(X, e, O, S_{\mathcal{R}}, \mathcal{K})) * \\
\quad\quad \textcircled{6}\mathsf{FlowImp}(X, e, T_{\mathsf{ob}}, T_{\mathsf{eco}}, T'_{\mathsf{ob}}, T'_{\mathsf{eco}}, S_{\mathsf{ob}}, S_{\mathsf{eco}}, \mathcal{K}, 0, \top) * \mathsf{wpb}^{\mathcal{K}}_{tid}\ s'\ \{Q\} \\
\mathit{Otherwise}:\ \\
\quad \ldots
\end{array}\right)
\end{array}\right)
\end{array}$$

Figure 6.18: Sketch of the definition of base weakest precondition. X is the execution graph; I is the instruction memory; e is the event identifier of the current event.

identical modulo a small number of abstract predicates that the instantiation of the model needs to provide. This symmetry is intentional, as it demonstrates the model's flexibility and generality, in particular that the treatment does not rely on any specific structure of any order. Using the subscript $\mathcal{R}$ for order-parametric predicate and variables, we write $\mathbin{*}_{\mathcal{R}}\, P_{\mathcal{R}}$ to denote $P_{\mathsf{ob}} * P_{\mathsf{eco}}$ in the definitions, and write $T_{\mathcal{R}}$ to refer to the two copies of T for the two orders in the text explanation to minimise redundancy.

As in AxSL, the fundamental role of the base weakest precondition is to track global maps for tied-to assertions and constrain their evolution. $T_{\mathcal{R}}$ at ❶ are the new level-indexed global tied-to maps of type $\mathit{Lvl} \to \mathit{Eid} \to \mathit{iProp}$ for the two orders. These maps are interpreted by the tied-to interpretation predicates $\mathsf{TI}_{\mathcal{R}}$ at ❷ and ❸, providing the authoritative view of all tied-to assertions. The predicates must adhere to basic agreement and update rules with the tied-to assertions. One can make "local" updates to the $T_{\mathcal{R}}$ maps around the current node e, resulting in $T'_{\mathcal{R}}$ (❹), to reflect the resource flow from certain nodes to e. These updates must adhere to the new flow implication ❻, ensuring that updates from $T_{\mathcal{R}}$ to $T'_{\mathcal{R}}$ are correct for levels between 0 and $\mathcal{K}$, with the resource $\top$ remaining (which effectively means 'nothing') at levels higher than $\mathcal{K}$. The so-called update level sets $S_{\mathcal{R}}$, satisfying the WFIf predicate, are also required after the tied-to map update; we defer their explanation, and that of the persistent order typing map O (⓫), until their motivations are introduced in Section 6.5.1.

$$\mathsf{WFIf}_{\mathcal{R}}(X, e, O, S, \mathcal{K}) \triangleq$$
$$\forall \mathcal{R}s = O[\mathsf{Loc}(e, X)].\ (\mathcal{R} \in \mathcal{R}s \to \mathsf{WF}_{\mathcal{R}}(S, \mathcal{K})) * (\mathcal{R} \notin \mathcal{R}s \to S = \emptyset)$$
$$\mathsf{FlowImp}(X, e, T_{\mathsf{ob}}, T_{\mathsf{eco}}, T'_{\mathsf{ob}}, T'_{\mathsf{eco}}, S_{\mathsf{ob}}, S_{\mathsf{eco}}, \mathcal{K}, k, R) \triangleq$$
$$(\mathcal{K} < k \to R) *$$
$$\left(\begin{array}{l} \mathcal{K} \geq k \to \\ \exists U_{\mathsf{ob}}, U_{\mathsf{eco}}. \\ \left(\ast_{\mathcal{R}}\ \mathsf{FlowImpAt}_{\mathcal{R}}(X, e, k, T_{\mathcal{R}}[k], T'_{\mathcal{R}}[k], S_{\mathcal{R}}, U_{\mathcal{R}})\right) * \\ \left((U_{\mathsf{ob}} * U_{\mathsf{eco}}) \mathrel{-\!\!*} \mathsf{FlowImp}(X, e, T_{\mathsf{ob}}, T_{\mathsf{eco}}, T'_{\mathsf{ob}}, T'_{\mathsf{eco}}, S_{\mathsf{ob}}, S_{\mathsf{eco}}, \mathcal{K}, k+1, R)\right) \end{array}\right)$$
$$\mathsf{FlowImpAt}_{\mathcal{R}}(X, e, k, \tau, \tau', S, U) \triangleq$$
$$\exists \tau_{in}, P.\ \mathsf{TiedUpd}_{\mathcal{R}}(X, e, \tau, \tau', \tau_{in}, P) * \mathsf{StateUpdAt}_{\mathcal{R}}(X, e, k, \tau_{in}, S, P, U)$$
$$\mathsf{StateUpdAt}_{\mathcal{R}}(X, e, k, \tau_{in}, S, P, U) \triangleq$$
$$\left(\ast_{(\_ \mapsto P_{in}) \in \tau_{in}} P_{in}\right) \mathrel{-\!\!*}$$
$$\left((k \in S) \mathrel{-\!\!*} \left(\forall M, \sigma.\ \textcircled{7} \left(\begin{array}{l} \mathsf{PredOf}(X.\mathcal{R}, e) \subseteq \mathrm{dom}(M) * (M[e] \ \#\ S_{\geq k}) * \\ (\forall e' \in \mathsf{PredOf}(X.\mathcal{R}, e).\ \mathsf{WF}_{\mathcal{R}}(M[e']_{\leq k}, k)) \end{array}\right) \mathrel{-\!\!*} \right.\right.$$
$$\left.\left. \textcircled{8}\mathsf{SI}_{\mathcal{R}}(X, \sigma, M) \Rrightarrow \exists \sigma'.\ \textcircled{9}\mathsf{SI}_{\mathcal{R}}(X, \sigma', M \cup_{\mathsf{p}} \{e \mapsto \{k\}\}) * P * U\right)\right)$$
$$\ast\textcircled{10}\ ((k \notin S) \Rrightarrow\ (P * U))$$

Figure 6.19: The the definition of level-indexed flow implication. The machinery controlling the state update order is highlighted in yellow. $S_{\geq k}$ is a subset of S that includes all levels of S that are greater than or equal to k, likewise to $S_{\leq k}$. $M \cup_{\mathsf{p}} \{e \mapsto \{k\}\}$ is a point-wise union that add the current level k to the update level set of the current event $M[e]$. $\mathsf{WF}_{\mathcal{R}}$ has to be monotone: $\forall S, k.\ \mathsf{WF}_{\mathcal{R}}(S_{\leq k+1}, k+1) \mathrel{-\!\!*} \mathsf{WF}_{\mathcal{R}}(S_{\leq k}, k)$.

Level-indexed flow implication FlowImp with states

The simplified version of FlowImp was detailed in Section 6.3; here, we focus on the difference and new components. The top-level definition of FlowImp in Figure 6.19 differs from the simplified version by incorporating the single-level flow implication $\mathsf{FlowImpAt}_{\mathcal{R}}$, which governs resource updates at the current event e and level k. The parameters of FlowImp differ slightly from its simplified version, taking $T_{\mathcal{R}}$ and $T'_{\mathcal{R}}$, the global tied-to maps before and after the resource update at event e. Additionally, the parameters $S_{\mathcal{R}}$ are passed directly to FlowImpAt.

Single-level flow implication $\mathsf{FlowImpAt}_{\mathcal{R}}$

The single-level flow implication $\mathsf{FlowImpAt}_{\mathcal{R}}$ in Figure 6.19 tracks resource flow for order $\mathcal{R}$ at a specific level. It extends the flow implication of AxSL for ob, with adjustments for level indexing. It constrains the *resource flow and updates* happening around e captured by the update of the per-level tied-to map τ of type $Eid \to iProp$

(which is exactly the type of the tied-to map of AxSL) to τ'. The predicate $\mathsf{TiedUpd}_{\mathcal{R}}$ adapted from AxSL ensures that this map update involves consuming τ_{in} from τ and attaching a new resource P to e. The resource update from τ_{in} to P is governed by $\mathsf{StateUpdAt}_{\mathcal{R}}$, which generalises its AxSL counterpart to support level-indexing. The update-enable sets $S_{\mathcal{R}}$ are again simply passed to $\mathsf{StateUpdAt}_{\mathcal{R}}$.

Flexible state update with $\mathsf{StateUpdAt}_{\mathcal{R}}$

$\mathsf{StateUpdAt}_{\mathcal{R}}(X, e, k, \tau_{in}, S, \ldots)$ in Figure 6.19 tracks resource updates at event e in level k. Its main advancement over its predecessor in AxSL is the introduction of logical states, which are crucial to implement non-atomic points-tos by tracking progress in the eco order. The predicate quantifies over a logical state σ, and links updates of $\mathcal{R}$-based resources to the update progress map M via a state interpretation $\mathsf{SI}_{\mathcal{R}}$, as in ❽ and ❾. Updates are limited to one event and level at a time, and a maximum of $\mathcal{K}$ updates can occur across events of an execution for order $\mathcal{R}$. These updates must adhere to a specific sequence for soundness. The map M associates events with their update level sets S, representing levels where state updates have been performed. For the current event e, if k is not in S (case ❿), the state update is skipped, and resources flow as in the simplified FlowImp. Otherwise, updates proceed if condition ❼ is met. Figure 6.20 depicts an illustration of this case. ❼ imposes constraints on M, which effectively controls when the update can be made. The first line says that this update at event e and level k must occur after updates of its predecessors and before updates of e at higher levels; The second line involves the new predicate $\mathsf{WF}_{\mathcal{R}}$ which requires that the sets of all e predecessors in M are well-formed up to the current level k. One application of this well-formedness condition is to enforce that updates at certain levels have to be performed, as we will see later in Section 6.5.2. This well-formedness predicate is also employed by $\mathsf{WPIf}_{\mathcal{R}}$ in the model of the base weakest precondition to selectively enable well-formedness of S. This depends on whether the order $\mathcal{R}$ is chosen for the event address (if any) in the order select map O, which maps locations to the set of orders enabled for reasoning. The map O is persistent (⓫) and user-determined. The integration of O, M, and S provides selective and flexible state updates, allowing users to decide when to update these order-based states by view-shifting the abstract interpretation predicates. Section 6.5.2 elaborates on how the AxSL+ instance in Section 6.4 leverages this flexibility to implement proof rules.

6.5.2 Instantiating the Model

To use the semantic model defined in the previous subsection, one must provide concrete implementations for the abstract predicates using ghost states. Specifically, these predicates include:

- interpretation GI for graph and instruction memory
- interpretation SI_{O} for the order typing map

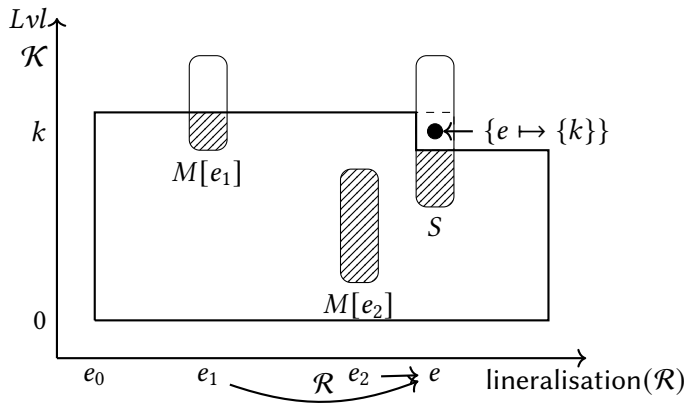


Figure 6.20: An illustration of the $k \in S$ case of $\mathsf{StateUpd}_{\mathcal{R}}(X, e, k, \tau_{in}, S, \cdots)$. The x-axis is a linearisation of the order $\mathcal{R}$. The y-axis is the levels starting from 0. e_1 and e_2 are the two $\mathcal{R}$-predecessors of e. The shapes for $M[e_1]$ and $M[e_2]$ include the state updates accumulated by their flow implications. The shaded parts of the shapes correspond to the updates that have been performed, and the blank parts correspond to the updates that have not been performed. The condition on M requires that it can only involve the shaded parts to perform the update at event e at level k (denoted by the dot on the top right). After the update, the small area of S containing the dot is also shaded, reflected by adding $e \mapsto \{k\}$ to M. Together, the conditions ensure that all the updates have to be performed from left to right, and from bottom to top.

- state interpretations $\mathsf{SI}_{\mathcal{R}}$
- well-formedness conditions for update level set $\mathsf{WF}_{\mathcal{R}}$
- interpretations for global tied-to maps $\mathsf{TI}_{\mathcal{R}}$

The instantiation, alongside with the model of the assertion language, are used to prove the soundness of the proof rules. In this subsection, we detail definitions for the logic instance AxSL+$^{\mathsf{NA}}$ of Section 6.4, highlighting derivable properties crucial to the soundness of the proof rules.

Protocols, $\mathsf{SI}_{\mathsf{ob}}$, and $\mathsf{WF}_{\mathsf{ob}}$

We define the type of the logical state σ_{ob} as $\mathit{Addr} \to (\mathit{Val} \to \mathit{Lvl} \to \mathit{Eid} \to \mathit{iProp})$, mapping locations to protocol predicates governing the corresponding location. The key idea behind this ob instantiation resembles that of AxSL: we require the protocol resource Φ to hold for all write events at all levels, as depicted in ❷ in Figure 6.21. This is enforced by the $\mathsf{WF}_{\mathsf{ob}}$ condition, ensuring the update level set S_{ob} contains all levels. Figure 6.22 illustrates the shape of M_{ob} enforced by $\mathsf{WF}_{\mathsf{ob}}$. To track protocols for locations, we employ the higher-order savedPred resource algebra (technically, Iris CMRA) to store them as ghost states, as shown in ❶. This setup ensures agreement between the state interpretation and the protocol assertion

(omitting the later modality required by step-indexing):

$$\mathsf{SI}_{\mathsf{ob}}(X, \sigma_{\mathsf{ob}}, M) * \mathsf{Prot}(x, \Phi) \vdash \sigma_{\mathsf{ob}}[x] = \Phi$$

This agreement allows concluding that Φ holds at all levels for the ob-before write when reading externally from x. Thus, the resources of the write, stored in the state interpretation, are accessible to the read.

$$\begin{aligned}
&\mathsf{SI}_{\mathsf{ob}}(X, \sigma_{\mathsf{ob}}, M) \triangleq \\
&\quad \Big(\exists g : \mathit{Addr} \to \mathit{Gname}.\ \boxed{\bullet g}^{\gamma_{\mathsf{ob}}} * \circledast_{(x \mapsto (\Phi;\gamma)) \in \sigma_{\mathsf{ob}}; g} \textcircled{1}\ \boxed{\mathsf{savedPred}(\Phi)}^{\gamma}\Big) * \\
&\quad \textcircled{2}\ (\forall (e \mapsto S) \in M.\ \forall x, v.\ X.\mathsf{E}[e] = \mathsf{W}\ x\ v \mathrel{-\!\!*} \circledast_{k \in S} \sigma_{\mathsf{ob}}[x](v, k, e)) \\
&\mathsf{Prot}(x, \Phi) \triangleq \exists \gamma.\ \boxed{\circ\{x \mapsto \gamma\}}^{\gamma_{\mathsf{ob}}} * \boxed{\mathsf{savedPred}(\Phi)}^{\gamma} \\
&\mathsf{WF}_{\mathsf{ob}}(S, k) \triangleq S = \{k' \mid 0 \le k' \le k\}
\end{aligned}$$

Figure 6.21: The instantiation of ob predicates and the model of the protocol assertion.

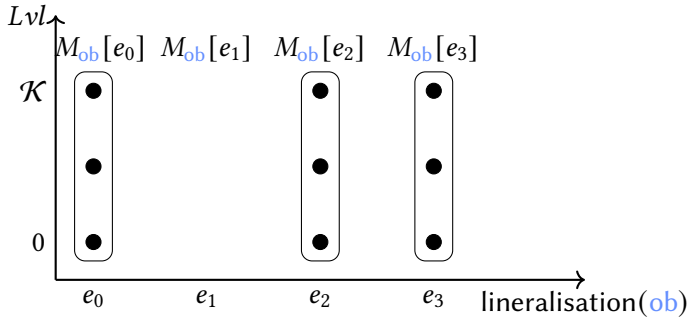


Figure 6.22: An illustration of the shape of M_{ob}. A dot indicates that there is a state update to apply at the corresponding event and level. It has to contain the state updates at all the levels from 0 to $\mathcal{K}$ for the events e_0, e_2, e_3 of the locations on which ob reasoning is enabled in O; and contains no updates (e.g. e_1) otherwise.

Non-atomic points-tos, $\mathsf{SI}_{\mathsf{eco}}$, and $\mathsf{WF}_{\mathsf{eco}}$

We define σ_{eco} as $\mathit{Addr} \to \mathit{List}(\mathit{Eid})$, mapping locations to their write history. ❷ in Figure 6.23 requires that the write list ℓ is sorted by co. The AgFrac RA in ❶ maintains two identical copies of ℓ, allowing updates when both are owned. One copy resides in the state interpretation, the other in the non-atomic points-to assertion. This assertion ensures that the latest write to x is of value v, by asserting that event e_w with value v is the last write in ℓ (❸). To prevent outdated reads, ❹ asserts that the latest write must be ob ∩ ext-before or po-before event e (if the events differ). Typically, e is set as the event tied to the assertion, ensuring ❹ holds when the

assertion is passed around locally and externally. Thus, reads from non-atomic locations can only access the last write of ℓ, and writes only append to ℓ by updating the assertion. The well-formedness condition for eco is weak: it only requires that at most one update can be performed for an event to give the illusion that levels do not exists, since the instantiation does not utilise the level information of M. Figure 6.24 depicts a shape of M_{eco} compatible with this WF_{eco} condition.

$$
\begin{aligned}
&\mathsf{SI}_{\text{eco}}(X, \sigma_{\text{eco}}, M) \triangleq \\
&\quad (\forall (x \mapsto \ell) \in \sigma_{\text{eco}}, \forall e \in \ell.\, e \in \mathrm{dom}(M)) * \\
&\quad \left(\exists g : \mathit{Addr} \to \mathit{Gname}.\ \boxed{\bullet g}^{\gamma_{\text{eco}}} * \mathbin{\text{✻}}_{(x \mapsto (\ell;\gamma)) \in \sigma_{\text{eco}}; g} \textcircled{1}\, \boxed{\mathsf{AgFrag}(\ell, 1/2)}^{\gamma}\right) * \\
&\quad \textcircled{2}\, (\forall e, e' \in \mathrm{dom}(M).\, \forall x = \mathsf{Loc}(e, X).\, e \mathrel{\mathsf{co}} e' \mathrel{-\!\!*} \exists i, j.\, \sigma[x][i] = e * \sigma[x][j] * i < j) \\
&x \overset{\text{NA}}{\mapsto} v_{[e]} \triangleq \\
&\quad \exists \gamma, \ell.\ \boxed{\circ\{x \mapsto \gamma\}}^{\gamma_{\text{eco}}} * \boxed{\mathsf{AgFrag}(\ell, 1/2)}^{\gamma} * \\
&\qquad \exists e_w.\, (e_w : \mathsf{W}\ x\ v) * \textcircled{3}(e_w = \mathsf{last}(\ell)) * \textcircled{4}\, ((e_w\ (\mathsf{ob} \cap \mathsf{ext})\ e) \vee (e_w\ \mathsf{po}\ e) \vee e = e_w) \\
&\mathsf{WF}_{\text{eco}}(S, \_) \triangleq \mathsf{size}(S) \leq 1
\end{aligned}
$$

Figure 6.23: The instantiation of eco predicates and the model of the non-atomic points-to assertion.

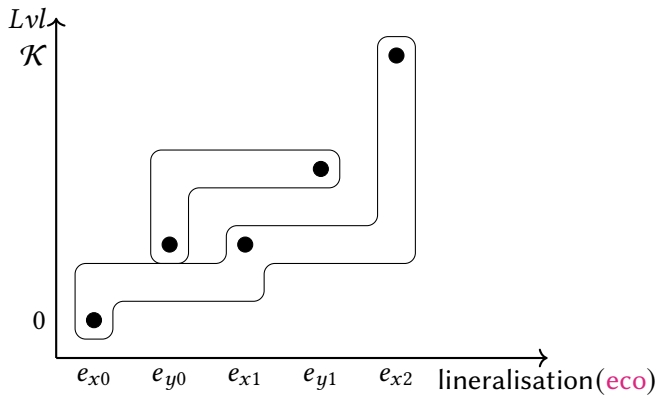


Figure 6.24: An illustration of the shape of M_{eco}. e_{x0}, e_{x1}, e_{x2} are events of x, and e_{y0}, e_{y1} are events of y. The two sub-shapes contain the state updates of the events of location x and y, assuming eco reasoning is enabled on the two locations according to O. Each of these sub-shapes contains only one state update per event.

6.5.3 Soundness of Proof Rules

AxSL+ rules are proven sound with respect to its semantic model and the implementation of the interpretation predicate presented in this section.

Theorem 6.5.1 (Soundness). *AxSL+ proof rules are sound.*

6.6 Adequacy

The statement of the adequacy theorem of AxSL+ builds on the one from AxSL, with minor changes to account for level-indexing. The proof of this theorem, however, diverges significantly from AxSL due to the introduction of level-indexed variables and terms, and the addition of flexible state updates. These factors introduce substantial complexity while retaining the two-phase structure of the original proof. Phase one involves constructing the level-indexed edge annotation and ensuring its well-formedness, which is more intricate. Phase two, instead of merely relying on induction over ob, requires three separate inductions: first on the level upper bound, and then once on each order.

This section outlines the necessary adaptations for the adequacy theorem, followed by a proof sketch that includes key intermediate steps and auxiliary lemmas.

Theorem 6.6.1 (Adequacy theorem of AxSL+). *For any initial thread states $\vec{C}$, meta-level propositions $\vec{\varphi}$ (one for each thread), valid execution graph X, and instruction memory I, we have*

$$\left(\bigwedge_{tid=1}^{n} \mathsf{Ctd}\ \vec{C}(tid) \xrightarrow{tid,X,I}_{\mathsf{h}}{}^{*} \mathsf{Done}\ \_\right) \Rightarrow$$
$$\left(\begin{array}{l}\exists ❶\mathcal{K}, \sigma_{\mathsf{ob}}, \sigma_{\mathsf{eco}}, O. \\ \vdash \dot{\Rightarrow} \left(\begin{array}{l} \exists M.\ ❷\mathsf{InitSt}_{\mathcal{R}}(X, \sigma_{\mathcal{R}}, M, O, \mathcal{K}) * \\ \left(*_{\mathcal{R}}\ ❸\mathsf{TI}_{\mathcal{R}}(\{(k,e) \mapsto \mathsf{Emp} \mid 0 \le k \le \mathcal{K} \wedge e \in \mathsf{InitNodes}(X)\})\right) * \\ ❹ \square\, \mathsf{GI}(\langle X, I\rangle) * ❺ \square\, \mathsf{SI}_{\mathsf{O}}(L) * \\ ❻ *_{tid=1}^{n} \left(\mathsf{wp}_{tid,\mathcal{K}}\ \mathsf{Ctd}\ \vec{C}(tid).p\ \{\lceil \vec{\varphi}(tid) \rceil\}\right) \end{array}\right) \end{array}\right) \Rightarrow$$
$$\left(\bigwedge_{tid=1}^{n} \vec{\varphi}(tid)\right)$$

where

$$\mathsf{InitSt}_{\mathcal{R}}(X, \sigma, M, O, \mathcal{K}) \triangleq ❼\mathsf{PgWF}_{\mathcal{R}}(X, M, O, \mathcal{K}, \mathsf{InitNodes}(X)) * ❽\mathsf{SI}_{\mathcal{R}}(X, \sigma, M)$$

and

$$\mathsf{PgWF}_{\mathcal{R}}(X, M, O, \mathcal{K}, D) \triangleq (\forall (e \mapsto S) \in M.\ \mathsf{WFIf}_{\mathcal{R}}(X, e, O, S, \mathcal{K})) * \mathsf{dom}(M) = D$$

Similarly to the adequacy of AxSL, the statement overall has three components: the first line assumes a terminating execution of the program, the next line assumes the verification of the program in AxSL+, and the last line is the conclusion that the pure facts $\vec{\varphi}$ proven in AxSL+ holds at the meta level. The theorem incorporates a concrete level upper bound $\mathcal{K}$ (❶), which acts as a key parameter. It also involves an order typing map L, which is a persistent map determining reasoning at each address, and which is logically interpreted as ❺. These two parameters vary between proofs, and thus needed to be provided by the user alongside the initial states $\sigma_{\mathcal{R}}$. The interpretations of initial states ❽ are required to be established in ❷, with a

consistent and well-formed initial update progress map M covering initial nodes as in ❼. Furthermore, initial resources may be allocated along with state interpretations. For each order $\mathcal{R}$, the theorem requires an interpretation of the initial tied-to map (❸), mapping initial events to 'empty' at all levels. ❹ and ❺ are the interpretations of persistent values, which are trivial to establish. Finally, the weakest preconditions of all threads (❻) are required, all parameterised by the same upper bound $\mathcal{K}$.

6.6.1 Overview of the Proof

The proof of adequacy of AxSL+ follows the two-phase strategy of AxSL. In phase one, the weakest preconditions are unfolded to collect flow implications along program order. Phase two then applies these implications in sequence along ob (and now eco, separately) to ensure sound resource transfer and state updates. Compared to AxSL, the level-indexed semantic model introduces significant complexity.

We recall the key aspects of the AxSL proof and dedicate subsequent subsections to detailed proof sketches for the revised phases.

In phase one of the AxSL adequacy proof, induction on the reduction trace of the program while keeping track of the tied-to map interpretation, allows unfolding the weakest preconditions of the threads. By pushing the tied-to interpretation through the view shift in the model of weakest preconditions, we track updates to the tied-to map based on flow implications. This process leads to the construction of an edge annotation map, capturing resource flow along ob edge between events, represented as a map of type $\mathit{Eid} \to \mathit{Eid} \to \mathit{iProp}$. By the end of this phase, the final tied-to map records which resources are associated with which events after resource flow has occurred. A new predicate establishes the relationship between the resources in the edge annotation map and those in the (node annotation) tied-to map, ensuring that if the resources indeed flow according to the edge annotation map, starting from the initial resources, then the resources of the tied-to maps indeed ties to the corresponding events.

Phase two of AxSL adequacy proof builds upon the results of phase one, replaying the resource updates and flows recorded by the edge annotation map, but this time along ob. By the end of this second induction, we obtain all the tied resources with which we show the pure postconditions of the weakest preconditions. The adequacy proof concludes by applying the Iris base adequacy theorem to extract postconditions to the meta logic.

6.6.2 Phase 1: Collecting Flow Implications

The goal of phase one of the adequacy proof of AxSL+ is similar to that of AxSL: collecting flow implications along program order, as formulated in Theorem 6.6.2. Given the hypotheses from the adequacy statement (on the left of the entailment), the proof of phase one begins with induction on the program reduction trace, unfolding the weakest preconditions while propagating the interpretations of the tied-to maps $T_{\mathcal{R}}$. Concurrently, it collects level-indexed flow implications FlowImp. The result is

the final tied-to maps $T_{\mathcal{R}}$ (❶), which are well-formed, satisfying the assertions in ❷. Importantly, this ensures that these maps include all non-initial events and levels up to the specified upper bound, so phase two does not get stuck because of a missing flow implication.

Two critical differences from the original phase one from AxSL emerge. First, during unfolding, we also collect the update level sets $S_{\mathcal{R}}$, and use them to construct global update progress maps $M_{\mathcal{R}}$ (❸) that contain the well-formed update level sets of all non-initial nodes (❹). Second, because the tied-to maps and the flow implications are now all level-indexed, we construct edge annotation maps in a more complicated way. Specifically, one edge annotation map is created for each order at each level. The new predicate FlowAnnot (❺) recursively relates these annotation maps to the tied-to maps. The edge annotation maps are existentially quantified and implicitly defined within FlowAnnot for brevity. This new predicate also records all the state updates allowed by the update progress maps $M_{\mathcal{R}}$. The final outcome of this phase is a separating implication, asserting that the pure post conditions ❼ holds given all tied resources at all events and levels in the final tied-to maps (❻).

Lemma 6.6.2 (Phase 1: Collecting by Unfolding)**.** *Given initial tied-to maps $T_{\mathcal{R}}$ and order typing map O,*

$$
\begin{array}{l}
\left(\bigwedge_{tid=1}^{n} \mathsf{Ctd}\ \vec{C}(tid) \xrightarrow{tid,X,I}_{\mathsf{h}}{}^{*} \mathsf{Done}\ \_\right) * \left(\mathbin{\circledast}_{tid=1}^{n} \mathsf{wpb}_{tid}^{\mathcal{K}}\ \mathsf{Ctd}\ \vec{C}(tid).p\ \{\lceil\vec{\varphi}(tid)\rceil\}\right) * \\
(\mathbin{\circledast}_{\mathcal{R}} \mathsf{TI}_{\mathcal{R}}(T_{\mathcal{R}}) * \mathsf{dom}(T_{\mathcal{R}}) = (\{k | 0 \le k \le \mathcal{K}\} \times \mathsf{InitNodes}(X))) * \\
\square\, \mathsf{GI}(\langle X, I\rangle) * \square\, \mathsf{SI}_{\mathsf{O}}(O) \\
\vdash \Rrightarrow \\
\exists ❶ T'_{\mathsf{ob}}, T'_{\mathsf{eco}}.\ \left(\mathbin{\circledast}_{\mathcal{R}} \mathsf{TI}_{\mathcal{R}}(T'_{\mathcal{R}}) * ❷\mathsf{TiedMapWF}(X, T'_{\mathcal{R}}, \mathcal{K})\right) * \\
\left(\begin{array}{l}\exists ❸ M'_{\mathsf{ob}}, M'_{\mathsf{eco}}.\ \left(❹ \mathbin{\circledast}_{\mathcal{R}} \mathsf{PgWF}_{\mathcal{R}}(X, M'_{\mathcal{R}}, O, \mathcal{K}, \mathsf{NInitNodes}(X))\right) * \\ \quad ❺\mathsf{FlowAnnot}(X, T'_{\mathsf{ob}}, T'_{\mathsf{eco}}, M'_{\mathsf{ob}}, M'_{\mathsf{eco}}, \mathcal{K}, 0)\end{array}\right) * \\
\left(\left(❻ \mathbin{\circledast}_{\mathcal{R}} \mathsf{PullTied}(T'_{\mathcal{R}})\right) \mathrel{-\!\!*} ❼ \bigwedge_{tid=1}^{n} \vec{\varphi}(tid)\right)
\end{array}
$$

where

$$\mathsf{PullTied}(T) \triangleq \mathbin{\circledast}_{e \in \mathsf{NInitNodes}(X)} \left(\mathbin{\circledast}_{k=0}^{\mathcal{K}} T[k][e]\right)$$

6.6.3 Phase 2: Applying Flow Implications

Phase two of the adequacy proof of AxSL+ replays the resource updates and flows recorded in the edge annotation maps of FlowAnnot (❹), while concurrently performing the optional state updates captured by the update progress maps $M_{\mathcal{R}}$. The auxiliary Theorem 6.6.3 serves as the cornerstone of phase two. It involves the final tied-to maps $T_{\mathcal{R}}$, the update progress maps $M_{\mathcal{R}}$, and their well-formedness assertions from phase one (❶ and ❷). Additionally, the lemma requires the initial states and their interpretations (❸) as provided in the adequacy statement hypothesis.

The conclusion of the lemma is behind an iterated update modality, enabling updates across all levels for all non-initial nodes (❺). The resulting final states $\sigma'_{\mathcal{R}}$ and their interpretations (❻) are consistent with the union of the update progress maps for both initial and non-initial events. This consistency ensures that all state updates recorded in FlowAnnot have been executed. Consequently, the tied resources in $T_{\mathcal{R}}$ are derived, allowing to conclude the pure postconditions established in Theorem 6.6.2. After showing this lemma, the final step of applying Iris base adequacy on the postconditions is standard.

Lemma 6.6.3 (Phase 2: Applying by Inductions)**.** *Given the final tied-to maps $T'_{\mathcal{R}}$, the level-enabled maps $M'_{\mathcal{R}}$, both collected from Theorem 6.6.2, and the initial states $\sigma_{\mathcal{R}}$ with consistent update progress maps $M_{\mathcal{R}}$,*

$$
\begin{aligned}
&\vdash \\
&\quad \left(❶ \circledast_{\mathcal{R}} \mathsf{PgWF}_{\mathcal{R}}(X, M'_{\mathcal{R}}, O, \mathsf{NInitNodes}(X)) * ❷\mathsf{TiedMapWF}(X, T'_{\mathcal{R}}, \mathcal{K})\right) \mathrel{-\!\!*} \\
&\quad \left(❸ \circledast_{\mathcal{R}} \mathsf{InitSt}_{\mathcal{R}}(X, \sigma_{\mathcal{R}}, M_{\mathcal{R}}, O, \mathcal{K})\right) \mathrel{-\!\!*} \\
&\quad \left(❹\mathsf{FlowAnnot}(X, T'_{\mathsf{ob}}, T'_{\mathsf{eco}}, M'_{\mathsf{ob}}, M'_{\mathsf{eco}}, \mathcal{K}, 0)\right) \mathrel{-\!\!*} \\
&\quad \left(\circledast_{\mathcal{R}} \dot{\Rightarrow}^{❺\mathcal{K}\times\mathsf{Size}(\mathsf{NInitNodes}(X))} \exists \sigma'_{\mathcal{R}}.\ ❻\mathsf{SI}_{\mathcal{R}}(X, \sigma'_{\mathcal{R}}, M_{\mathcal{R}} \cup M'_{\mathcal{R}}) * ❼\mathsf{PullTied}(T'_{\mathcal{R}})\right)
\end{aligned}
$$

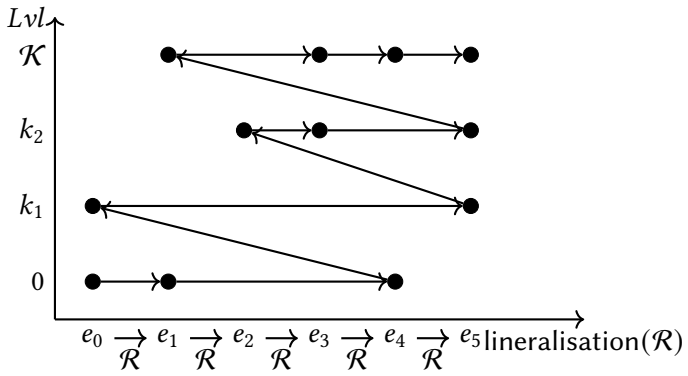


Figure 6.25: An illustration of the application of state updates in the adequacy proof for order $\mathcal{R}$. The dots represent the state updates collected by the progress update map M at every node and level. The adequacy proof performs all of these updates in the (lexicographic) order indicated by the arrows.

The key insight of the proof of Theorem 6.6.3 lies in handling dependencies between updates systematically: higher-level updates rely on the completion of lower-level updates, and within each level, the update for an event depends on updates of its predecessors. This is achieved through induction on the upper bound $\mathcal{K}$, with nested inductions on the two orders. The update sequence is visualised in Figure 6.25, illustrating the application sequence of state updates captured in $M'_{\mathcal{R}}$.

6.7 Further Steps

AxSL+$^{\mathsf{NA}}$ of Section 6.4 demonstrates minimal mixed-order reasoning with non-atomics and level-indexed ob protocol. In this section, we discuss several directions for extending AxSL+$^{\mathsf{NA}}$, including those that can be directly implemented based on the semantic model presented in Section 6.5, and those that we believe require more work.

6.7.1 Basic Coherence Reasoning for Racy Locations

One limitation of the proof rules presented in Section 6.4 is that we can only reason about coherence for non-racy locations. We can lift this limitation by defining a history predicate $\mathsf{Coh}(x, \ell)$ that exposes the history of write ℓ for location x, which allows us to conduct basic coherence reasoning in the presence of data races. The model of Section 6.5 is fully compatible with this new assertion. One only needs to tweak the implementation of the eco state interpretation $\mathsf{SI}_{\mathsf{eco}}$ so that it captures more information about coherence. In the rest of this subsection, we present properties from which the derivation of the proof rules is straightforward. We can have two variants of the predicate to handle the case of write-write races, or only write-read races. For the former, the rules we could obtain with this history predicate is rather weak:

HIST-AGREE
$$\mathsf{SI}_{\mathsf{eco}}(X, \sigma, M) * \mathsf{Coh}(x, \ell) \vdash \exists \ell'.\, \ell +\!\!+ \ell' = \sigma[x]$$

HIST-UPDATE
$$\frac{e : \mathsf{W}\ x\ v \qquad \exists \ell_1, \ell_2.\, \mathsf{last}(\ell_1)\ \mathsf{eco}\ e * \ell_1 +\!\!+ \ell_2 = \sigma[x]}{\mathsf{SI}_{\mathsf{eco}}(X, \sigma, M) \Rrightarrow \exists \ell_2' = \mathsf{insert}(\ell_2, e).\, \mathsf{SI}_{\mathsf{eco}}(X, \sigma[x \mapsto (\ell_1 +\!\!+ \ell_2')], M \cup_{\mathsf{p}} \{e \mapsto \{k\}\})}$$

HIST-DUPLICATE
$$\frac{x \in \mathsf{dom}(\sigma)}{\mathsf{SI}_{\mathsf{eco}}(X, \sigma, M) \Rrightarrow \mathsf{SI}_{\mathsf{eco}}(X, \sigma, M) * \mathsf{Coh}(x, \sigma[x])}$$

HIST-PERSIST
$$\mathsf{Coh}(x, \ell) \Rrightarrow \mathsf{Coh}(x, \ell) * \mathsf{Coh}(x, \ell)$$

The implementation of the $\mathsf{Coh}(x, \ell)$ predicate holds a fragmental view ℓ of the history of x, while the eco state interpretation $\mathsf{SI}_{\mathsf{eco}}$ holds the full view (of which ℓ is a prefix).

For locations with only write-read races, we assume there is only a single writer, which holds the full and exclusive history with $\mathsf{Coh}(x, \ell)_W$. All the readers may still own prefixes of the history with $\mathsf{Coh}(x, \ell)_R$, for which the same weak rules above

holds. However, we also have the following strong rules for the writer:

HIST-SW-AGREE
$$\mathsf{SI}_{\mathsf{eco}}(X, \sigma, M) * \mathsf{Coh}(x, \ell)_W \vdash \ell = \sigma[x]$$

HIST-SW-UPDATE
$$\frac{e : \mathsf{W}\ x\ v \qquad \mathsf{last}(\sigma[x])\ \mathsf{eco}\ e}{\mathsf{SI}_{\mathsf{eco}}(X, \sigma, M) * \mathsf{Coh}(x, \sigma[x])_W \Rrightarrow \mathsf{SI}_{\mathsf{eco}}(X, \sigma[x \mapsto (\sigma[x] +\!\!+ [e])], M \cup_{\mathsf{p}} \{e \mapsto \{k\}\}) * \mathsf{Coh}(x, (\sigma[x] +\!\!+ [e]))_W}$$

HIST-SW-DUPLICATE
$$\mathsf{Coh}(x, \ell)_W \Rrightarrow \mathsf{Coh}(x, \ell)_W * \mathsf{Coh}(x, \ell)_R$$

HIST-SW-COMBINE
$$\mathsf{Coh}(x, \ell)_W * \mathsf{Coh}(x, \ell')_R \vdash \exists \ell''.\ \ell = \ell' +\!\!+ \ell''$$

The implementation of $\mathsf{Coh}(x, \ell)_W$ holds half of the ghost state of the full view of ℓ, while $\mathsf{Coh}(x, \ell)_R$ holds fragmental views as before. The former can also be used to implement non-atomic points-to assertions.

6.7.2 eco Protocols

The $\mathsf{Prot}(x, \Phi)$ assertion in AxSL+ allows resources to be transferred between threads along ob edges, facilitating local reasoning. Similarly, one can introduce a protocol assertion to enable resource transfer along eco edges for a given location.

This can be implemented by enforcing the protocol predicates on all the writes in the history. However, this protocol is stateless, meaning extra machinery must be implemented to distinguish between two writes of the same value in the history when sending resources. We conjecture that stateful eco-based protocols could make resource transfer more flexible in AxSL+.

Per-location protocols in the GPS logic are an example of such protocols. They have been shown to be robust and easy to use for verification of realistic programs. However, porting per-location protocols to AxSL+ presents a challenging task, particularly when combined with level-indexing. We believe that implementing them with tractable proof rules for resource transfer is difficult using the semantic model of Section 6.5 as-is.

In the remainder of this subsection, we will first introduce GPS protocols, and then elaborate on the challenges that arise when trying to integrate them into AxSL+.

GPS Per-location protocols

The per-location protocol predicate $\boxed{x : s \mid \tau}$ in GPS associates an abstract state s (from a state transition system, STS) and a predicate τ with location x, ensuring $\tau(v, s)$ holds for the value v at location x. This predicate abstracts away much reasoning of the acyclic condition on eco, exposing only the reasoning of the monotonicity of state transitions to the proof rules. For instance, the GPS write rule for atomics

ensures progress in the STS (*i.e.,* the protocol) in a thread-local manner: to move from a potentially outdated state s to s'', one only needs to show that for all $s' \sqsupseteq s$ in the STS, s'' is reachable.

iGPS refines GPS with three per-location protocol variants: persistent protocols (corresponding to the original protocols of GPS), single-writer protocols, and fractional protocols, each designed for specific scenarios, with corresponding proof rules of varying strength. We now elaborate on the first two.

Persistent protocols Persistent protocols, the most generic protocol variant from GPS, have relatively weak reasoning rules for writes. Here, each thread can own a copy of the protocol to track visible progress locally. This allows arbitrary data races on the location, but the protocol does not track a globally latest state on its own. More precisely, $\boxed{x : s \mid \tau}$ signifies that the location is *at least* at state s from the owning thread's perspective. During reads, the state s may update to $s' \sqsupseteq s$, granting resource $\tau(v, s')$ for the read outcome v, resulting in $\boxed{x : s' \mid \tau}$. This update reflects potential read-write races: since other threads may have made the protocol τ progress to s' with some writes, we might read from a later s' according to coherence. The rule for writes is even more restricted. This is because the writer thread has no information on how other threads have already moved the protocol forward, so it can only update the protocol state conservatively, taking into account how other concurrent writes may advance the state. In conclusion, persistent protocols supports both read-write and write-write races at the cost of restricted proof rules (in particular for writes).

Single-writer protocols Single-writer protocols support stronger proof rules by precluding reasoning about write-write races, requiring at most one writer, as the name suggests. Single-writer protocols use two assertions: $\boxed{x : s \mid \tau}_R$ is a reader protocol, which can be seen as a (potentially outdated) snapshot of the state transition: it asserts that the latest state of τ is at least s. The read rule with this persistent assertion is identical to the one with persistent protocols (and therefore accounts for read-write races). $\boxed{x : s \mid \tau}_W$ is a writer protocol, which can be seen as the full view: it asserts that the latest state of τ is exactly s. This assertion is non-duplicable, granting exclusive write permission to advance τ to the holding thread. This exclusivity enables freely updating τ's state with a stronger write rule.

Challenge in integrating GPS-style protocols

Integrating GPS-style protocols into AxSL+ is a challenging task. The introduction of level-indexing complicates the justification of sound resource transfer when using GPS-style protocols. To understand why this integration is difficult, and why we claim that tractable proof rules for GPS protocols cannot be easily implemented with the current semantic model, we examine the challenges that one faces when attempting to do so.

In this attempt, we develop an implementation of GPS-style protocols and sketch proof rules, and show how they suffer from usability issues. We expect that these issues would be shared by any approach, as we suspect that the root cause lies in the semantic model.

Figure 6.26 illustrates two specialised proof rules that we develop for the single-writer variant of GPS-style protocols. Assertion $\boxed{x : s \mid \tau, \xi}$ represents the GPS-style protocol in AxSL+, which additionally includes a function ξ mapping abstract states to levels. The level $\xi(s)$ for a state s corresponds to the level at which $\tau(v, s, e)$ is satisfied at event e. While this solution is somewhat limiting, it offers a simple way to track the level(s) at which resources for state s are satisfied.

Tracking this information is crucial for ensuring the soundness of the rules, as resource transfer must never flow to lower levels. This means that when receiving resources for state s, we must ensure that they are tied to the read event at a level that is not smaller than $\xi(s)$. This requirement is reflected in the proof rules in Figure 6.26, which we now discuss in detail.

HT-STR-REL-PERLOC-SW

$$\left\{\begin{array}{l} \mathsf{PoPred}(e_{\mathrm{po}}) * \textcircled{1}\left(\mathbin{\text{✻}}_{((k,e)\mapsto P)\in m}(e^{k}_{\mathrm{ob}} \hookrightarrow P)\right) * \\ \textcircled{2} e_0 \overset{k_0}{\underset{\mathrm{eco}}{\hookrightarrow}} \left(\exists s.\, \boxed{y : s \mid \tau, \xi}_W(e_0) * \psi(s)\right) * \\ (\forall s.\, \psi(s) \mathrel{-\!\!*} s \sqsubseteq s' * \textcircled{3} \mathcal{K} \geq \xi(s') \geq \max(K+1, k_0)) * \\ \forall e, v, e_w, v_w.\, \left(\begin{array}{l} \mathsf{GraphFacts}(e, x, v, e_w, v_w, e_{\mathrm{po}}) \mathrel{-\!\!*} \\ \left(\begin{array}{l} \mathsf{Lob}(\mathrm{dom}(m), e, K) * (e_0 \text{ po-loc } e) * \\ \textcircled{4}\left((\mathbin{\text{✻}}_{(\_\mapsto P)\in m} P) \mathrel{-\!\!*} \forall s.\, \psi(s) \mathrel{-\!\!*} \tau(v_w, s, e_w) \Rrightarrow \tau(v, s', e)\right) \end{array}\right) \end{array}\right) \end{array}\right\}$$

$$\mathsf{str}\ [y]\ v$$

$$\left\{\ \exists e.\, \mathsf{PoPred}(e) * \mathsf{GraphFacts}(e, x, v, e_{\mathrm{po}}) * \textcircled{5} e^{\xi(s')}_{\mathrm{eco}} \hookrightarrow \left(\boxed{y : s' \mid \tau, \xi}_W(e)\right)\ \right\}_{tid, \mathcal{K}}$$

HT-LDR-RLX-PERLOC-SW

$$\left\{\begin{array}{l} \mathsf{PoPred}(e_{\mathrm{po}}) * \textcircled{6} e_0 \overset{k_0}{\underset{\mathrm{eco}}{\hookrightarrow}} \left(\exists s.\, \boxed{y : s \mid \tau, \xi}_R(e_0) * \psi(s)\right) * \textcircled{7}(\mathcal{K} \geq k' \geq k_0) * \\ \forall e, v, e_w.\, \left(\begin{array}{l} \mathsf{GraphFacts}(e, x, v, e_w, e_{\mathrm{po}}) \mathrel{-\!\!*} \\ (e_0 \text{ po-loc } e) * \\ \textcircled{8}\left(\begin{array}{l} \forall s'.\, (\exists s.\, \psi(s) * s \sqsubseteq s') \mathrel{-\!\!*} \\ \quad \tau(v, s', e_w) \Rrightarrow \tau(v, s', e_w) * Q(v, s', e_w) \end{array}\right) \end{array}\right) \end{array}\right\}$$

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

$$\left\{\begin{array}{l} \exists e, v, e_w.\, \mathsf{PoPred}(e) * \mathsf{GraphFacts}(e, x, v, e_w, e_{\mathrm{po}}) * \\ e^{k'}_{\mathrm{eco}} \hookrightarrow \left(\begin{array}{l} \exists s'.\, (\forall s, \psi(s) \mathrel{-\!\!*} s \sqsubseteq s') * \\ \quad \textcircled{9} \boxed{y : s' \mid \tau, \xi}_R(e) * (\textcircled{10} k' \geq \xi(s') \mathrel{-\!\!*} \textcircled{11} Q(v, s', e_w)) \end{array}\right) \end{array}\right\}_{tid, \mathcal{K}}$$

Figure 6.26: Hypothetical sound proof rules with single-writer per-location protocols in AxSL+.

HT-STR-REL-PERLOC-SW is a rule for stores where the writer protocol is at some state s that satisfies $\psi(s)$. The protocol is currently tied to some po-loc-before event e_0 at

level k_0, as seen in ❷. This rule allows us to advance the protocol to some later state $s' \sqsupseteq s$. The rule loosely follows the shape of the original GPS rule, and is as strong as expected. When showing the protocol resources at s' in ❹, we can use resources tied to lob-before events (as shown in ❶). ❸ is the crucial level condition ensuring soundness: it requires that K, the maximum level of the resources flowing in, plus one order-switching, is not greater than the predetermined level of s'. In other words, the resources for transfer cannot flow to lower levels. It also requires that $\xi(s')$ is not smaller than k_0, ensuring that the protocol assertion itself cannot flow to lower levels. Intuitively, this condition ensures that the assertions and resources needed for this resource transfer are available at $\xi(s')$, the level where the transfer is happening. This is why we have the updated protocol tied at level $\xi(s')$ in ❺.

HT-LDR-RLX-PERLOC-SW is a rule for loads where the reader protocol is at state s, tied to some e_0 at level k_0. This rule allows us to receive resources from some later state s', as in ❽. In the postcondition, we specify that the received resources ⓫ along with the updated protocol ❾ are tied to the read event at some user-selected level k'. This is where the usability issue arises (the rule we give is also weaker than the corresponding GPS rule, which includes a choosable P in the precondition, which slightly alleviates but does not solve the usability problem we highlight). The actual resources received are conditioned by ❿, meaning that they are only available if the user-determined k' is not smaller than $\xi(s')$, the level where the resources are sent. This is problematic, as it renders the received resources barely usable. Indeed, the only way to discharge this condition is when we are sure that we are at the final state s_{fin}, in which case we can set $k' = \xi(s_{fin})$. This condition is otherwise unavoidable, since the user of the rule cannot pick k' based on the new state s', as s' is quantified *in* the tied-to assertion.

This problem stems from the semantic model. When taking a step, the model of the weakest precondition first requires the updated tied-to maps $T'_{\mathcal{R}}$ and then shows that the update satisfies the flow implication. The state interpretation of eco, from which we can know the updated protocol state s', is only accessible within the flow implication, meaning it is only available *after* selecting $T'_{\mathcal{R}}$.

Looking at this issue more broadly, it is questionable whether a semantic model without this problem even exists. Generally speaking, the desired proof rules seem to require solving a circular dependency between the protocol states and levels.

Although we do not know how to completely resolve this issue, there may be solutions that mitigate the impact of this extra condition. This is a key direction to explore, as we believe that using GPS-style protocols for resource transfer can significantly improve the scalability of AxSL+.

6.7.3 Modular Specifications for Concurrent Libraries

Another limitation of the current extension that we would like to address is the fixed upper bound, which limits compositionality. The fixed upper bound only works well when reasoning about a closed program. However, this is problematic when writing down modular specifications for concurrent libraries because it is not possible to

determine in advance the number of levels needed by clients for order-switching. Therefore, we would like to make this upper bound more flexible, by, for instance, allowing it to be mutable. This would enable the combination of two specifications by unifying their upper bounds. We anticipate that achieving this requires an even more flexible semantic model.

6.8 Conclusion

The separation of coherence (eco) and synchronisation order (ob) of Arm-A poses a challenge on sound and thread-modular reasoning of their combination. This makes well-established coherence-based abstractions not immediately available in program logics for Arm-A like AxSL that can reason about only ob. In this paper, we present the idea of level-indexing as an extension to AxSL to support mixing reasoning about ob and eco together. This allows us to capture key coherence patterns. We build a simple AxSL+$^{\mathsf{NA}}$ based on this idea, in which we construct a notion of non-atomic points-to to ease reasoning about non-racy locations. The generic semantic model AxSL+ is based only on acyclicity of orders, rather than any specific structure of ob or eco. Thus, we believe that this approach can be applied not only to extensions of the Arm-A model like virtual memory, but also to other memory model with multiple acyclic orders.

6.8.1 Future Work

In future work, we aim to mechanise the results and address several key areas of improvement for our approach. First, we will implement the plan outlined in Section 6.7, which is expected to enhance the robustness of our framework by addressing the identified constraints and limitations. Second, we will validate the logic with more diverse examples, including additional litmus tests and an approximate counter [McKenney, 2024, §5], which provides a more realistic example. Finally, we will explore the implementation of FSL-style modalities [Doko and Vafeiadis, 2016], further broadening the expressiveness of our logic.

An alternative phrasing of coherence An alternative and equivalent formulation of coherence is to forbid exactly five patterns in the execution graphs [Alglave, 2010]. It is interesting to investigate how to build coherence-based abstractions with this formulation, and compare it with the formulation used in this paper.

6.8.2 Related Work

All related work of this paper has been discussed in the AxSL papers [Hammond et al., 2024; Liu et al., 2024].

Bibliography

2021. *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE. doi:10.1109/SP40001.2021

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 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. 9035)*, Christel Baier and Cesare Tinelli (Eds.). Springer, 353–367. doi:10.1007/978-3-662-46681-0_28

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 150:1–150:29. doi:10.1145/3360576

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless Model Checking for POWER. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 134–156. doi:10.1007/978-3-319-41540-6_8

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 135:1–135:29. doi:10.1145/3276505

Jade Alglave. 2010. *A shared memory poetics*. Ph. D. Dissertation. L'université Paris Denis Diderot. http://www0.cs.ucl.ac.uk/staff/j.alglave/these.pdf

Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models, See Castagna and Gordon [2017], 3–18. doi: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. doi: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. doi: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. doi: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. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 40. doi:10.1145/2594291.2594347

Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 247–256. doi:10.1109/LICS.2001.932501

Andrew W. Appel. 2012. Verified Software Toolchain. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7226)*, Alwyn Goodloe and Suzette Person (Eds.). Springer, 2. doi:10.1007/978-3-642-28891-3_2

Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. In *Philosophical Transactions of the Royal Society A*, Vol. 375. Issue 2104. doi:10.1098/rsta.2016.0331

Arm. 2021. Morello project. Retrieved July 6, 2021 from https://www.morello-project.org/

Arm Ltd. 2022. *Arm Firmware Framework for Arm A-profile version 1.1 - DEN0077A*. Technical Report. https://documentation-service.arm.com/static/624d5f52dc9d4f0e74a54e5f

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, See Silva and Leino [2021], 303–316. doi:10.1007/978-3-030-81685-8_14

Rini Banerjee, Kayvan Memarian, Dhruv Makwana, Christopher Pulte, Neel Krishnaswami, and Peter Sewell. 2025. Fulminate: Testing CN Separation-Logic Specifications in C. *Proc. ACM Program. Lang.* 9, POPL, Article 43 (Jan. 2025), 33 pages. doi:10.1145/3704879

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics, See Vitek [2015], 283–307. doi: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. doi: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. doi:10.1145/1926385.1926394

Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. 2022. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture, See Sergey [2022], 174–203. doi:10.1007/978-3-030-99336-8_7

Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. 2016. A high assurance virtualization platform for ARMv8. In *European Conference on Networks and Communications, EuCNC 2016, Athens, Greece, June 27-30, 2016*. IEEE, 210–214. doi:10.1109/EUCNC.2016.7561034

Christoph Baumann, Oliver Schwarz, and Mads Dam. 2019. On the verification of system-level information flow properties for virtualized execution platforms. *J. Cryptogr. Eng.* 9, 3 (2019), 243–261. doi:10.1007/S13389-019-00216-4

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

Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson. 2022. View-Based Owicki-Gries Reasoning for Persistent x86-TSO, See Sergey [2022], 234–261. doi:10.1007/978-3-030-99336-8_9

Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). 2011. *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Lecture Notes in Computer Science, Vol. 6617. Springer. doi:10.1007/978-3-642-20398-5

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. doi:10.1145/1375581.1375591

Richard Bornat, Jade Alglave, and Matthew J. Parkinson. 2015a. 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

Richard Bornat, Jade Alglave, and Matthew J. Parkinson. 2015b. 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

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic, See Palsberg and Abadi [2005], 259–270. doi:10.1145/1040305.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 55–72. doi:10.1007/3-540-44898-5_4

Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 227–270. doi:10.1016/J.TCS.2006.12.034

Hongxu Cai, Zhong Shao, and Alexander Vaynberg. 2007. Certified self-modifying code. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 66–77. doi:10.1145/1250734.1250743

Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs, See Bobaru et al. [2011], 459–465. doi:10.1007/978-3-642-20398-5_33

Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-based Addressing. In *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, Forest Baskett and Douglas W. Clark (Eds.). ACM Press, 319–327. doi:10.1145/195473.195579

Giuseppe Castagna and Andrew D. Gordon (Eds.). 2017. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM. doi:10.1145/3009837

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 243–258. doi:10.1145/3341301.3359632

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

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

Arthur Charguéraud. 2023. *A Modern Eye on Separation Logic for Sequential Programs*. Habilitation à diriger des recherches. Université de Strasbourg. https://inria.hal.science/tel-04076725

Vijay Chidambaram. 2018. We found a bug in a verified file system! Twitter. https://twitter.com/vj_chidambaram/status/1047505696533741568

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. doi:10.1145/3519939.3523718

Edmund M Clarke. 1997. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*. Springer, 54–56. doi:10.1007/BFb0058022

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. 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, 23–42. doi:10.1007/978-3-642-03359-9_2

Karl Crary and Michael J. Sullivan. 2015. A Calculus for Relaxed Memory, See Rajamani and Walker [2015], 623–636. doi:10.1145/2676726.2676984

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 207–231. doi:10.1007/978-3-662-44202-9_9

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. doi: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. doi:10.1145/3519939.3523451

Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2020. Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12079)*, Armin Biere and David Parker (Eds.). Springer, 378–382. doi:10.1007/978-3-030-45237-7_24

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

Will Deacon. 2020. Virtualisation for the Masses: Exposing KVM on Android. http://linux-kernel.uio.no/pub/linux/kernel/people/will/slides/kvmforum-2020-edited.pdf

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 147–162. doi:10.1109/EUROSP.2016.22

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs, See Giacobazzi and Cousot [2013], 287–300. doi: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. doi:10.1007/978-3-642-14107-2_24

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 363–377. doi:10.1007/978-3-642-00590-9_26

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, See Jobstmann and Leino [2016], 413–430. doi:10.1007/978-3-662-49122-5_20

Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++, See Yang [2017], 448–475. doi: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. doi:10.1145/3453483.3454065

Xinyu Feng. 2009. Local rely-guarantee reasoning. 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, 315–327. doi:10.1145/1480881.1480922

Xinyu Feng and Zhong Shao. 2005. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 254–267. doi:10.1145/1086365.1086399

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software, See Palsberg and Abadi [2005], 110–121. doi:10.1145/1040305.1040315

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf

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. doi:10.1145/3009837.3009839

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2024. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 71, 1 (2024), 3:1–3:59. doi:10.1145/3623510

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. doi:10.1145/3434287

Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. doi:10.1145/3527318

Kourosh Gharachorloo. 1995. *Memory Consistency Models for Shared-Memory Multiprocessors.* Ph. D. Dissertation. Stanford University. doi:10.5555/891506

Roberto Giacobazzi and Radhia Cousot (Eds.). 2013. *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013.* ACM. doi:10.1145/2429069

Patrice Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and Pierre Wolper. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag, Berlin, Heidelberg.

Google LLC. 2021. pKVM. https://android-kvm.googlesource.com/linux/+/refs/heads/pkvm/

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. doi:10.1145/2830772.2830775

Hafnium development team. 2022. Hafnium — A security-focussed type-1 hypervisor. https://opensource.google/projects/hafnium

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 (2024), 604–637. doi: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. doi:10.1109/PDP.2016.103

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. doi:10.1145/78969.78972

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

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

Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6536)*, Raja Natarajan and Adegboyega K. Ojo (Eds.). Springer, 55–75. doi:10.1007/978-3-642-19056-8_4

Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). 2013. *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*. ACM. doi:10.1145/2509136

Bart Jacobs. 2014. *Verifying TSO Programs (Report CW660)*. Technical Report. https://lirias.kuleuven.be/bitstream/123456789/452373/1/CW660.pdf

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java, See Bobaru et al. [2011], 41–55. doi:10.1007/978-3-642-20398-5_4

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. doi:10.1145/2933575.2934536

Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The leaky semicolon: compositional semantic dependencies for

relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. doi:10.1145/3498716

Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. 2013. High-level separation logic for low-level code, See Giacobazzi and Cousot [2013], 301–314. doi:10.1145/2429069.2429105

Ranjit Jhala and Isil Dillig (Eds.). 2022. *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. ACM. doi:10.1145/3519939

Barbara Jobstmann and K. Rustan M. Leino (Eds.). 2016. *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. Springer. doi:10.1007/978-3-662-49122-5

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. doi:10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 256–269. doi:10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi: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, See Rajamani and Walker [2015], 637–650. doi: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. doi: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, See Castagna and Gordon [2017], 175–189. doi:10.1145/3009837.3009850

Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. 2013. Coq: the world's best macro assembler?. In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, Ricardo Peña and Tom Schrijvers (Eds.). ACM, 13–24. doi:10.1145/2505879.2505897

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. doi:10.1145/1743546.1743574

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1 (2014), 2:1–2:70. doi:10.1145/2560537

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel.. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 207–220. doi:10.1145/1629575.1629596

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. doi:10.1109/ICPP.1993.15

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (2018), 17:1–17:32. doi:10.1145/3158105

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. 2022. Truly stateless, optimal dynamic partial order reduction. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. doi:10.1145/3498711

Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. 2023. Unblocking Dynamic Partial Order Reduction. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13964)*, Constantin Enea and Akash Lal (Eds.). Springer, 230–250. doi:10.1007/978-3-031-37706-8_12

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries, See McKinley and Fisher [2019], 96–110. doi:10.1145/3314221.3314609

Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1157–1171. doi:10.1145/3373376.3378480 ASPLOS 2020 was canceled because of COVID-19..

Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models, See Silva and Leino [2021], 427–440. doi:10.1007/978-3-030-81685-8_20

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. doi:10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic, See Yang [2017], 696–723. doi:10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic, See Castagna and Gordon [2017], 205–217. doi:10.1145/3009837.3009855

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. doi: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. doi:10.1145/3062341.3062352

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

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. doi:10.1109/TC.1979.1675439

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. doi:10.1145/3527325

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. doi:10.1145/3385412.3386010

Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. 2024. Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 304–332. doi:10.1145/3689722

Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5850)*, Ana Cavalcanti and Dennis Dams (Eds.). Springer, 806–809. doi:10.1007/978-3-642-05089-3_51

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 3953–3970. https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei

Zongyuan Liu, Lars Birkedal, and Jean Pichon-Pharabod. 2025. First Steps Towards AxSL+. (2025).

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. (2024).

Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023a. Supplementary material: Coq development of VMSL. doi:10.5281/zenodo.7685774

Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023b. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1438–1462. doi: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

William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 148–174. doi:10.1145/3632848

Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. 2025. Model Checking C/C++ with Mixed-Size Accesses. *Proc. ACM Program. Lang.* 9, POPL (2025), 2232–2252. doi:10.1145/3704911

Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code, See Jhala and Dillig [2022], 841–856. doi:10.1145/3519939.3523704

Paul E. McKenney. 2024. Is Parallel Programming Hard, And, If So, What Can You Do About It? https://doi.org/10.48550/arXiv.1701.00854

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

Kathryn S. McKinley and Kathleen Fisher (Eds.). 2019. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM. doi:10.1145/3314221

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: a concurrent separation logic for multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 96:1–96:29. doi:10.1145/3408978

F. Lockwood Morris and Cliff B. Jones. 1984. An Early Program Proof by Alan Turing. *IEEE Ann. Hist. Comput.* 6, 2 (1984), 139–143. doi:10.1109/MAHC.1984.10017

Peter Müller (Ed.). 2020. *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. Springer. doi:10.1007/978-3-030-44914-8

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning, See Jobstmann and Leino [2016], 41–62. doi:10.1007/978-3-662-49122-5_2

Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 415–429. doi:10.1109/SP.2013.35

Magnus O. Myreen. 2009. *Formal verification of machine-code programs*. Ph. D. Dissertation. University of Cambridge. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-765.pdf

Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. 2007. Hoare Logic for ARM Machine Code. In *International Symposium on Fundamentals of Software Engineering, International Symposium, FSEN 2007, Tehran, Iran, April 17-19, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4767)*, Farhad Arbab and Marjan Sirjani (Eds.). Springer, 272–286. doi:10.1007/978-3-540-75698-9_18

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, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 568–582. doi:10.1007/978-3-540-71209-1_44

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 2008, Portland, Oregon, USA, 17-20 November 2008*, Alessandro Cimatti and Robert B. Jones (Eds.). IEEE, 1–8. doi:10.1109/FMCAD.2008.ECP.24

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources, See Shao [2014], 290–310. doi:10.1007/978-3-642-54833-8_16

Peter Naur. 1966. Proof of algorithms by general snapshots. *BIT* 6, 4 (July 1966), 310–316. doi:10.1007/BF01966091

Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. 2008. Runtime Checking for Separation Logic. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 4905)*, Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck (Eds.). Springer, 203–217. doi:10.1007/978-3-540-78163-9_19

Zhaozhong Ni and Zhong Shao. 2006. Certified assembly programming with embedded code pointers. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina,*

USA, January 11-13, 2006, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 320–333. doi:10.1145/1111037.1111066

Zhaozhong Ni, Dachuan Yu, and Zhong Shao. 2007. Using XCAP to Certify Realistic Systems Code: Machine Context Management. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4732)*, Klaus Schneider and Jens Brandt (Eds.). Springer, 189–206. doi:10.1007/978-3-540-74591-4_15

Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony C. J. Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1003–1020. doi:10.1109/SP40000.2020.00055

Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics, See Hosking et al. [2013], 131–150. doi:10.1145/2509136.2509514

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. doi:10.1016/J.TCS.2006.12.035

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. doi: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. doi: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. doi:10.1007/BF00268134

Jens Palsberg and Martín Abadi (Eds.). 2005. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM. doi:10.1145/1040305

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *Proceedings of the 34th ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 297–302. doi:10.1145/1190216.1190261

Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency, See Müller [2020], 599–625. doi:10.1007/978-3-030-44914-8_22

Thibaut Pérami, Zongyuan Liu, Nils Lauermann, Brian Campbell, Alasdair Armstrong, Thomas Bauereiss, and Peter Sewell. 2024. Reusable Rocq semantics of modern relaxed architectures. (2024).

Quentin Perret. 2020. Protected KVM: Memory protection of KVM guests in Android. https://linuxplumbersconf.org/event/7/contributions/780/

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. doi:10.1145/2837614.2837616

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 72:1–72:30. doi:10.1145/3290385

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. doi:10.1145/3158107

Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL (2023), 1–32. doi:10.1145/3571194

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, See McKinley and Fisher [2019], 1–15. doi:10.1145/3314221.3314624

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020. Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 151:1–151:28. doi:10.1145/3428219

Sriram K. Rajamani and David Walker (Eds.). 2015. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM. doi:10.1145/2676726

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1096–1120. doi:10.1145/3591265

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. doi: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. doi: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, See Jhala and Dillig [2022], 825–840. doi: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. doi: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. doi:10.1145/1993498.1993520

Susmit Sarkar, Peter Sewell, Luc Maranget, Shaked Flur, Christopher Pulte, Jon French, Ben Simner, Scott Owens, Pankaj Pawan, Francesco Zappa Nardelli, Sela Mador-Haim, Dominic Mulligan, Ohad Kammar, Jean Pichon-Pharabod, Gabriel Kerneis, Alasdair Armstrong, Thomas Bauereiss, and Jeehoon Kang. 2010–2024. RMEM: Executable operational concurrency model exploration tool for ARMv8, RISC-V, Power, and x86. [web interface].

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. doi:10.1145/1480881.1480929

Ilya Sergey (Ed.). 2022. *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. Springer. doi:10.1007/978-3-030-99336-8

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015a. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 77–87. doi:10.1145/2737924.2737964

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015b. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity, See Vitek [2015], 333–358. doi:10.1007/978-3-662-46669-8_14

Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. 2016. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 92–110. doi:10.1145/2983990.2983999

Thomas Sewell, Simon Winwood, Peter Gammie, Toby C. Murray, June Andronick, and Gerwin Klein. 2011. seL4 Enforces Integrity. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6898)*, Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer, 325–340. doi:10.1007/978-3-642-22863-6_24

Zhong Shao (Ed.). 2014. *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. Springer. doi:10.1007/978-3-642-54833-8

Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. 2015. A Separation Logic for Fictional Sequential Consistency, See Vitek [2015], 736–761. doi:10.1007/978-3-662-46669-8_30

Alexandra Silva and K. Rustan M. Leino (Eds.). 2021. *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Lecture Notes in Computer Science, Vol. 12759. Springer. doi:10.1007/978-3-030-81685-8

Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A, See Sergey [2022], 143–173. doi: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, See Müller [2020], 626–655. doi:10.1007/978-3-030-44914-8_23

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.* 3, POPL (2019), 19:1–19:28. doi:10.1145/3290332

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates, See Shao [2014], 149–168. doi:10.1007/978-3-642-54833-8_9

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. 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, 169–188. doi:10.1007/978-3-642-37036-6_11

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. doi:10.1007/978-3-319-89884-1_13

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. doi:10.1145/3133913

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. doi: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. doi:10.1145/2660193.2660243

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency, See Hosking et al. [2013], 867–884. doi:10.1145/2509136.2509532

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4703)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.). Springer, 256–271. doi:10.1007/978-3-540-74407-8_18

Simon Friis Vindum and Lars Birkedal. 2023. Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 632–657. doi:10.1145/3622820

Jan Vitek (Ed.). 2015. *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. Springer. doi:10.1007/978-3-662-46669-8

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

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. 2019. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. Technical Report UCAM-CL-TR-927. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.html

Ian Wehrman. 2012. *Weak-Memory Local Reasoning*. Ph. D. Dissertation. University of Texas at Austin. http://hdl.handle.net/2152/19475

Ian Wehrman and Josh Berdine. 2011. A proposal for weak-memory local reasoning. In *Low-level languages and applications (LOLA)*. https://www.microsoft.com/en-us/research/publication/a-proposal-for-weak-memory-local-reasoning/

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. doi:10.1007/978-3-642-37036-6_12

M. V. Wilkes and R. M. Needham. 1979. *The Cambridge CAP Computer and Its Operating System*. Elsevier. https://www.microsoft.com/en-us/research/publication/the-cambridge-cap-computer-and-its-operating-system/

Hongseok Yang (Ed.). 2017. *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. Springer. doi:10.1007/978-3-662-54434-1

Nobuko Yoshida and Lorenzo Gheri. 2020. A Very Gentle Introduction to Multiparty Session Types. In *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 11969)*, Dang Van Hung and Meenakshi D'Souza (Eds.). Springer, 73–93. doi:10.1007/978-3-030-36987-3_5

Dachuan Yu and Zhong Shao. 2004. Verification of safety properties for concurrent assembly code. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, Chris Okasaki and Kathleen Fisher (Eds.). ACM, 175–188. doi:10.1145/1016850.1016875

Yang Zhang and Xinyu Feng. 2014. Program Logic for Local Reasoning in TSO. (2014).