Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement

AMIN TIMANY, Aarhus University, Denmark
SIMON ODDERSHEDE GREGERSEN, Aarhus University, Denmark
LÉO STEFANESCO, MPI-SWS, Germany
JONAS KASTBERG HINRICHSEN, Aarhus University, Denmark
LÉON GONDELMAN, Aarhus University, Denmark
ABEL NIETO, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

Expressive state-of-the-art separation logics rely on step-indexing to model semantically complex features and to support modular reasoning about imperative higher-order concurrent and distributed programs. Step-indexing comes, however, with an inherent cost: it restricts the adequacy theorem of program logics to a fairly simple class of safety properties.

In this paper, we explore if and how *intensional refinement* is a viable methodology for strengthening higher-order concurrent (and distributed) separation logic to prove non-trivial safety and liveness properties. Specifically, we introduce Trillium, a language-agnostic separation logic framework for showing intensional refinement relations between *traces* of a program and a model. We instantiate Trillium with a concurrent language and develop Fairis, a concurrent separation logic, that we use to show liveness properties of concurrent programs under fair scheduling assumptions through a fair liveness-preserving refinement of a model. We also instantiate Trillium with a distributed language and obtain an extension of Aneris, a distributed separation logic, which we use to show refinement relations between distributed systems and TLA$^+$ models.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Higher order logic**; **Separation logic**; **Hoare logic**; **Programming logic**; *Invariants*; *Program verification*; *Program specifications*; *Pre- and post-conditions*; **Abstraction**.

Additional Key Words and Phrases: Distributed systems, separation logic, refinement, higher-order logic, concurrency, formal verification

1 INTRODUCTION

There is a tension between the *expressivity* of program logics and how much they say about the semantics of the programs being verified, that is, the strength of their *adequacy theorems*. As program

---

Authors' addresses: Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Simon Oddershede Gregersen, Aarhus University, Denmark, gregersen@cs.au.dk; Léo Stefanesco, MPI-SWS, Germany, lstefane@mpi-sws.org; Jonas Kastberg Hinrichsen, Aarhus University, Denmark, hinrichsen@cs.au.dk; Léon Gondelman, Aarhus University, Denmark, gondelman@cs.au.dk; Abel Nieto, Aarhus University, Denmark, abeln@cs.au.dk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

logics become more expressive—to handle sophisticated programming language features such as higher-order functions and references, and to support modular and general library specifications—they require increasingly complex semantic models to justify them.

In this work, we consider Iris [Jung et al. 2015], state of the art in terms of expressivity: in particular, it supports higher-order quantification, nested Hoare triples, higher-order ghost state, and *impredicative* invariants, i.e., invariants that can contain any Iris proposition, including invariants. Through its adequacy theorem, the Iris program logic [Jung et al. 2018] is designed to prove three kinds of properties about programs: (1) postconditions, *i.e.*, properties of the final values computed, (2) progress, *i.e.*, the program never gets stuck, and (3) preservation of invariants, *i.e.*, all invariants stated by the user hold throughout execution. Note how these properties form a particular class of *safety* properties.[1] That is—as trace properties—these are somewhat trivial, in that they are all of the form: "for any s such that $s_0 \to^* s$, we have $P(s)$" for some property P, where $\to^*$, is the reflexive-transitive closure of the operational semantics of the program. In particular, this class of properties does not include liveness properties or non-trivial safety trace properties like: "the value of the counter must increase monotonically without skipping over any number." While proving non-trivial safety trace properties is not a fundamental limitation of the Iris logic,[2] the lack of support for liveness properties is. This inherent limitation is related to the fact that Iris's semantic model relies on step-indexing, which is crucial for the soundness of impredicative invariants. This fact is a compromise that the designers of Iris have made. On the one hand, making the semantic model step-indexed, and thereby enabling impredicative invariants, allows many important applications. Notably, it allows us to construct logical relations models for proving type safety and contextual equivalences for expressive programming languages, *e.g.*, System F with recursive types, higher-order references, and concurrency [Krebbers et al. 2017]. On the other hand, step-indexing inherently restricts program logics to only be able to express properties that *concern* finite prefixes of program execution. This effectively dooms any program logic developed on top of the Iris base logic from supporting liveness properties—at least directly. The thesis that we explore in this work is that *refinement is a viable methodology for strengthening higher-order concurrent (and distributed) separation logic to prove non-trivial safety and liveness trace properties.*

Focus and Methodology. In this paper, we use Iris to establish *intensional refinements* between programs and labeled transition systems (LTSs), including the strong notion of *liveness-preserving* refinements, for *concurrent and distributed programs*. We develop Trillium, a language-agnostic generic program logic, whose adequacy theorem guarantees the existence of a refinement between the program and an LTS chosen by the prover. This is in addition to the usual properties enjoyed by program logics for safety reasoning as mentioned above, *i.e.*, postconditions, progress, and preservation of invariants. The key insight is that, by showing an *intensional* refinement between a program and an LTS, we can—indirectly—establish non-trivial safety trace properties and liveness properties such as fair termination of concurrent programs. By proving that the LTS enjoys the property of interest (which is often, if not always, simpler than proving it for the program itself), we can use the refinement relation to "transport" the property to the program. For this reason, we will refer to the LTS as a *model* or *specification* of a program *implementation*.

---

[1]Recall that a *safety* property is a property that expresses "nothing bad ever happens throughout execution", as opposed to *liveness* properties which express that "something good will eventually happen".
[2]For instance, Birkedal et al. [2021] do use an Iris program logic to prove some limited intensional safety trace properties, *e.g.*, "a file can only be accessed if it has previously been opened and not subsequently closed."

Contributions. In summary, we make the following contributions:

- We introduce Trillium (§3), a language-agnostic separation logic framework for establishing intensional refinement relations between traces of program executions and models.
- We develop Fairis (§4), a higher-order concurrent separation logic for showing liveness properties of concurrent programs under fair scheduling assumptions through a fair termination-preserving refinement of a carefully chosen model (see discussion in §2).
- We showcase Fairis on a number of challenging but small concurrent examples (§4.2).
- We instantiate Trillium with AnerisLang to get an extension of Aneris [Krogh-Jespersen et al. 2020] that can be used to show intensional refinements of distributed systems (§5).
- We use the Aneris instantiation of Trillium to show that two distributed protocols, two-phase commit and single-decree Paxos, refine their abstract TLA⁺ [Lamport 1992] specifications. To the best of our knowledge, this is the first foundationally verified proof that a concrete implementation of a distributed protocol correctly implements an abstract TLA⁺ specification.
- We further show functional correctness and strong eventual consistency of a concrete implementation of a Conflict-Free Replicated Data Type (CRDT). The challenging part is incorporating the notion of fairness of the inter-replica communication; if messages from one replica are just ignored, then eventual consistency will never be reached. Moreover, the concurrent interactions with the user-exposed operations makes it non-trivial to reason about eventual consistency. To the best of our knowledge, this is the first such proof that takes into account the inter-replica communication at the level of the implementation. For the sake of space, we have relegated further details about this example to the accompanying appendix.
- All the results that appear in the paper have been formalized in the Coq proof assistant using the Iris separation logic framework.

2 BACKGROUND AND KEY OBSERVATIONS

We will think of the operational semantics of a concurrent program as an LTS where the transition labels are thread identifiers corresponding to the thread taking the step. That an LTS refines another is a standard notion: two states are in a refinement relation if there exists a *forward simulation relation R* that relates them (see, *e.g.*, Cleaveland and Sokolsky [2001]).[3] The goal in this paper is to transport *intensional* safety and liveness properties of (possibly infinite) *traces* along such a refinement relation, *e.g.*, transporting the property "the value of the counter increases (or stays the same) monotonically without skipping over any number" from the model Chain in Figure 1 to the program count_up in Figure 2.

To transport intensional properties, we will work with *intensional refinement*, which is a lockstep relation where every step of the program is matched by a step of the model. This is, of course, too strong if taken literally: for example, the step of computation corresponding to a recursive call of count_up does not increment the counter and hence does not correspond to a step in the model. For now, we will ignore this issue; in the following section we will present constructions on LTSs that will allow us to relax the correspondence between the program and the model, while still allowing intensional properties to be transported.

We recall the precise definitions of forward simulation and refinement. The definitions are relative to a parameter ξ, a relation on traces, which provides for a bit of flexibility, by allowing one to restrict attention to traces satisfying ξ.

---

[3]Cleaveland and Sokolsky [2001] work with relations on a single LTS whereas we work with two LTSs. Nonetheless, all the results carry over straightforwardly by simply considering an LTS that is the disjoint union of the two LTSs we consider.
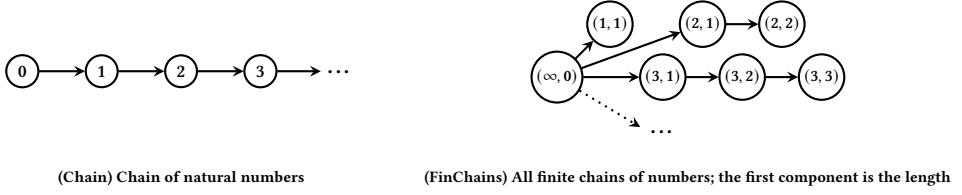
(Chain) Chain of natural numbers　　　　　　**(FinChains) All finite chains of numbers; the first component is the length**

Fig. 1. Two simple LTSs representing the *infinite* chain of natural numbers Chain, and all *finite* chains of natural numbers FinChains.

```
let rec count_up l = FAA l 1; count_up l in count_up l
```

Fig. 2. The program count_up. Here FAA is the (atomic) fetch-and-add operation which increments the integer stored in its first argument (a reference) by the given amount in the second argument. We assume that the value of l is zero at the beginning.

Definition 2.1 (ξ-forward simulation). Let ξ be a binary relation on finite traces. A relation R is a *ξ-forward simulation*, written $ForSim_\xi(R)$, if:

$$ForSim_\xi(R) \triangleq \big(\forall \tau, \tau'. \, R(\tau, \tau') \implies \xi(\tau, \tau')\big) \wedge$$
$$\big(\forall \tau, \tau', l, s. \, R(\tau, \tau') \wedge last(\tau) \xrightarrow{l} s \implies \exists l', s'. \, R(\tau \xrightarrow{l} s, \tau' \xrightarrow{l'} s')\big)$$

where *last* maps a trace to its end state.

Definition 2.2 (Intensional refinement). Let ξ be a binary relation on finite traces. A finite trace τ is an *intensional refinement* of a finite trace τ', with respect to parameter ξ, written $\tau \leqslant_\xi \tau'$, if there exists a ξ-forward simulation relation R such that $R(\tau, \tau')$. That is,

$$\tau \leqslant_\xi \tau' \triangleq \exists R. \, ForSim_\xi(R) \wedge R(\tau, \tau')$$

In our running example, the parameter ξ is used to restrict our attention to traces where the value of the counter and the model state agree at all times.

The Trillium program logic is designed to establish (*cf.* Theorem 3.2) an intensional refinement $c \leqslant_\xi m$ between singleton traces consisting of a program state c and a model state m. From Lemma 2.4 below it then follows directly that any possibly-infinite execution of the program can be matched by a possibly-infinite trace of the model in such a way that all their corresponding finite prefixes are in the intensional refinement relation.

Definition 2.3 (Trace Relation Extension). Given a relation R on finite traces, we lift R to possibly-infinite traces, written $\hat{R}$, by considering all finite prefixes as follows:

$$\hat{R}(\tau_1, \tau_2) \triangleq \forall \tau_1', \tau_2'. \, |\tau_1'| = |\tau_2'| \wedge prefix(\tau_1', \tau_1) \wedge prefix(\tau_2', \tau_2) \implies R(\tau_1', \tau_2')$$

where $prefix(\tau, \tau')$ means τ is a prefix of τ'.

LEMMA 2.4 (INTENSIONAL REFINEMENT, LIFTING). *Let L_1 and L_2 be two LTSs, and ξ a relation on traces between these LTSs. Let $s_1 \in L_1$ and $s_2 \in L_2$ be two states such that $s_1 \leqslant_\xi s_2$ (seen as singleton traces). For all possibly-infinite traces τ_1 of L_1 such that $first(\tau_1) = s_1$ there exists a possibly-infinite trace τ_2 where $first(\tau_2) = s_2$ such that $\tau_1 \hat{\leqslant}_\xi \tau_2$. Here, first maps traces to their initial state.*

2.1 Program Steps That Do Not Correspond to Steps in the Model

The core idea of the methodology we propose in this paper is that the model refined by the program is more abstract, and hence simpler and easier to reason about, than the program. Thus, in general, there will be steps in the program that do not correspond to any step in the model. One way to reconcile this with our notion of refinement is to allow for *stuttering*. That is, allow the program to take a step while the model stays in the same state. We will support stuttering by *lifting* the model into an LTS that allows stuttering. We consider two kinds of lifting, one that is only sound for intensional safety trace properties, and one that is sound for liveness trace properties as well. For the former, we show that it is sound in the sense that, if a program refines the *lifted* model, and the *original* model enjoys an intensional safety trace property, so does the program. Similarly, the latter lifting is sound with respect to liveness trace properties.

We remark on a subtle point here, namely that Trillium is a framework, which means that when we combine the base program logic of Trillium with one of the two aforementioned liftings, we obtain two different program logics. The choice of lifting presents a compromise between expressivity and simplicity of the *derived* logical principles of the program logic. In particular, for the lifting that is sound for intensional safety properties, we obtain a program logic that is conservative with respect to the ordinary program logic of Iris in that all the reasoning principles of the Iris program logic are still sound, and in addition one obtains simple reasoning principles that allow proving refinements (*cf.* §5). For the lifting that is sound with respect to liveness properties, we obtain a program logic that is more involved but allows proving liveness properties (*cf.* §4).

Stutter-Lifting: Sound for Intensional Safety Trace Properties. This lifting is very simple: it essentially amounts to adding *self-loops*, with a special label, to all states of the LTS. For example, the LTS Chain from Figure 1 would result in the following LTS, where dotted arrows are added to support stuttering:
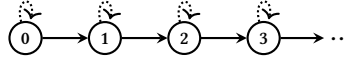
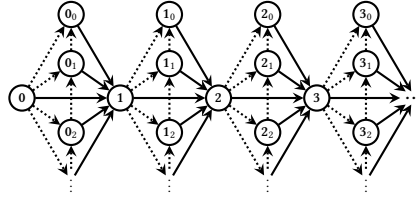
(Chain-Loops) Stutter-Lifting of the LTS Chain in Figure 1

The program count_up in Figure 2 is an intensional refinement of the model Chain-Loops above if we take the parameter ξ to relate the value of memory location l with the state of the model—the recursive call then corresponds to taking the self-loop in the relevant state. As mentioned above, this stutter-lifting is sound for intensional safety trace properties. Hence, to show that "the value of the counter increases (or stays the same) monotonically without skipping over any number", it suffices to show that this property is enjoyed by Chain in Figure 1.

However, this lifting is not sound for liveness properties, because it is also refined by (using the same ξ parameter) the following program:

```
let rec loop () = loop () in FAA l 1; loop ()
```

which only increments l once and afterwards loops forever. To see this, take a very simple liveness property like "the value of the counter is eventually 3" which is trivially true for the LTS Chain in Figure 1, but not for the program above. The culprit here is unrestricted stuttering.

Fin-Stutter-Lifting: also Sound for Liveness Properties. To obtain soundness with respect to liveness properties, we define a fin-stutter-lifting construction, which only allows for finite stuttering. That is, instead of adding loops, it essentially adds finite unrollings of loops, by creating copies of each state, each of which allows at most a certain, fixed number of stuttering steps. For instance, the Fin-Stutter-Lifting of the LTS Chain in Figure 1 is given blow:

(Chain-Fin-Stutter) Fin-Stutter-Lifting of the LTS Chain in Figure 1

The idea is to add states $\{n_i \mid i \in \mathbb{N}\}$ (for each state n), all of which intuitively correspond to state n. From a state n, we can either go to the state $n + 1$, or *stutter* to a state n_s from where we can at most stutter s times before going to state $n + 1$. The Fin-Stutter-Lifting construction is sound for liveness properties. In fact, it is the core idea of the so-called fuel construction we present in §4.3. Note that, as expected, the program above which increments the counter only once and then loops forever does not refine the LTS Chain-Fin-Stutter.

2.2 Step-Indexing and Finite Approximability

As mentioned in §1, step-indexing prevents direct reasoning about liveness properties because it restricts reasoning to finite prefixes of program execution. The same complication arises when establishing refinement relations. In this work, this issue shows up in Theorem 3.2 as the *relative image-finiteness* side-condition that we will discuss and formally define here. In order to see this issue concretely, revisit the LTS FinChains from Figure 1. FinChains has no infinite paths and hence any program that refines the fin-stutter-lifting of FinChains, regardless of the ξ parameter, must terminate. However, in a step-indexed logic, one can show a refinement relation between the program count_up in Figure 2 and the fin-stutter-lifting of FinChains. To see this, simply take the ξ parameter for this step-indexed refinement relation to require that the value of l corresponds to the second component of the state in the fin-stutter-lifting of FinChains. In this case, for any finite trace of the program, there is a trace in the fin-stutter-lifting of FinChains that matches it according to this ξ relation. The key point here is that when the value of l goes from 0 to 1 in the program, on the model side we go from the state $(\infty, 0)$ to $(n, 1)$, where n is the number of steps of execution being considered. Hence, it would not be sound, *in this case*, to conclude an intensional refinement relation from the refinement relation established in the step-indexed logic. The crux of the issue here is the unbounded choice of transitions going out of the state $(\infty, 0)$ which allows us to pick a path based on the number of steps of execution that we are considering. Below we analyze this problem more formally, and conclude that a so-called *relative image-finiteness* side-condition suffices to circumvent the problem. (Note that the problem of unbounded branching is already present in the fin-stutter-lifting construction; we will discuss this issue further when we explain the fuel construction in §4.3.)

We first define a notion of *finite approximation*, which intuitively corresponds to what *guarded recursive predicates* compute. Guarded recursive predicates are those defined as fixed points using the step-indexing technique, *e.g.*, the weakest preconditions underlying the program logic of both Iris and Trillium.

Definition 2.5. Let F be a function on the space of binary relations on traces of LTSs L_1 and L_2, *i.e.*, $F : 2^{tr(L_1) \times tr(L_2)} \rightarrow 2^{tr(L_1) \times tr(L_2)}$, where 2^A is the powerset of A, and $tr(L)$ is the set of all traces of the LTS L. We define the *finite approximation* of F, written $FinApprox(F)$, as

$$FinApprox(F) \triangleq \bigcap_{i=0}^{\infty} R_F^i \qquad \text{where } R_F^0 \triangleq tr(L_1) \times tr(L_2) \text{ and } R_F^{i+1} \triangleq F(R_F^i)$$

The upshot of the limitation of step-indexed logics is that the best we can hope to conclude from a refinement relation defined in a step-indexed logic is a finite approximation of the refinement relation. Specifically, in our case we can conclude $FinApprox(F_{\leqslant_\xi})$, where $F_{\leqslant_\xi}$ is the function whose greatest fixed point (by the Knaster-Tarski fixed point theorem) is the intensional refinement relation in Definition 2.2:

$$F_{\leqslant_\xi}(R) = \{(\tau, \tau') | \forall l, s.\ last(\tau) \xrightarrow{l} s \implies \exists l', s'.\ R(\tau \xrightarrow{l} s, \tau' \xrightarrow{l'} s')\} \cap \xi$$

In other words, we can only conclude $\leqslant_\xi$ if $\leqslant_\xi$ is finitely approximable, *i.e.*, if $\leqslant_\xi = FinApprox(F_{\leqslant_\xi})$, which is well-known not to be the case in general for refinement relations. Indeed, one frequently used condition for finite approximation is so-called image-finiteness [Cleaveland and Sokolsky 2001, Thm. 2.6]. An LTS is said to be image-finite if, for any state s and label l, there exists only finitely many states s' such that $s \xrightarrow{l} s'$. As we will discuss in §4.2.2, however, it is desirable to consider LTSs with infinite branching. For this reason we relax the image-finiteness condition by considering the weaker *relative image-finiteness* condition (a property of the ξ relation, not the LTS), defined below. For relative image-finiteness, it is sufficient that for any state s of the program and for any transition $s \xrightarrow{l} s'$, there are only finitely many LTS transitions that correspond to $s \xrightarrow{l} s'$ allowed by the ξ relation.

Definition 2.6 (Relative image-finiteness). Let ξ be a relation on traces of two LTSs. The relation ξ is *relatively image-finite* if, for any τ, τ', and transition $last(\tau) \xrightarrow{l} s$, the following set is finite:

$$\{(l', s') \mid \xi(\tau \xrightarrow{l} s, \tau' \xrightarrow{l'} s')\}$$

THEOREM 2.7 (FINITE APPROXIMATION). *Let ξ be a relatively image-finite relation on traces of two LTSs. The intensional refinement relation $\leqslant_\xi$ is finitely approximable, i.e., $\leqslant_\xi = FinApprox(F_{\leqslant_\xi})$.*

2.3 Further Discussions

Coming up with the Appropriate Model. One natural question regarding the methodology that we present in this paper is "how does one come up with the appropriate model and the parameter ξ for the verification task at hand?" We argue that coming up with the appropriate model and ξ parameter is of the same nature, and indeed part of picking the appropriate specification, *e.g.*, relevant preconditions and postconditions. Hence, there is no obvious, one-size-fits-all answer. Indeed, the model and the relation often need to be designed so as to facilitate establishing the safety trace property or liveness property of the program we wish to prove, *e.g.*, the relation ξ and the model Chain in Figure 1 that we chose for the program count_up in Figure 2, in order to establish that "the value of the counter increases monotonically without skipping over any number". Moreover, at a technical level, the limitation of relative-image-finiteness of the ξ relation restricts us in the choice of the model and the ξ parameter. For these reasons, the chosen model may not be arbitrarily abstract. It must reflect some of the core characteristics of the program, at least to the extent necessary for the trace property in question, and for the ξ to be relative image-finite. It is our hypothesis that coming up with appropriate models and ξ relations is feasible in most, if not all, interesting examples. However, in the present paper we only present the foundation and methodology of using intensional refinement to strengthen the expressivity of step-indexed higher-order concurrent (and distributed) separation logics, and support it with simple examples. A proper experimental evaluation of the hypothesis, using a wide range of more advanced examples, is beyond the scope of the current work, and we leave it for future work.

What about Transfinite Iris? Transfinite Iris [Spies et al. 2021] is a variant of Iris whose model is step-indexed over an arbitrary ordinal (as opposed to the natural numbers used in the model of ordinary Iris). The upshot of this change is that Transfinite Iris satisfies the so-called "existential

property" [Spies et al. 2021] which in effect renders the (relative) image-finiteness side-condition unnecessary. We believe that the work of this paper could also be carried out on top of Transfinite Iris, dispensing with the relative image-finiteness side-condition, albeit at the cost of other possible complications in proofs.[4] Spies et al. [2021] prove termination and termination-preserving refinements of sequential programs but do not show any (preservation of) liveness properties beyond termination nor do they treat concurrent programs.

What about the Approaches for Contextual Refinement? There have been several works on establishing *contextual* refinement in Iris for complex sequential and concurrent programming languages [Frumin et al. 2018, 2020; Georges et al. 2022; Gregersen et al. 2023; Jacobs et al. 2022, 2021; Krebbers et al. 2017; Krogh-Jespersen et al. 2017; Timany and Birkedal 2019; Timany et al. 2018]. In essence, contextual refinement boils down to showing that if one program terminates, so should the other. In these previous works, it is established by the use of invariants in Iris. This implies that the approach only allows one to show that for any finite prefix of execution of the first program (the implementation side) there exists a finite execution of the second program (the specification side) and the final states correspond. This notion of refinement is too weak for our purposes: (1) it says nothing about infinite executions and hence does not help us establish liveness properties, and (2) it does not allow us to transfer non-trivial safety trace properties. In the Appendix we discuss an illustrative example.

3 TRILLIUM: A TRACE PROGRAM LOGIC FRAMEWORK

In this section, we give a more detailed account of the general Trillium logic and a formal statement of its adequacy theorem. We first detail how we instrument the operational semantics of the domain programming language with "locale" transition identifiers (essentially thread id's), to facilitate thread-level properties such as fair scheduling. Second, we present the fundamentals of the Trillium logic, and its adequacy theorem (Theorem 3.2). We focus on the novelties of Trillium but will recount necessary constructions of the Iris base logic briefly at a high level.

Language Agnostic Framework. The Trillium program logic is language agnostic and is defined with respect to any programming language which comes with an operational semantics given by a notion of expression $e \in Expr$, value $v \in Val \subseteq Expr$, evaluation context $K \in Ectx$, program state $\sigma \in State$ (a model of, *e.g.*, the heap and/or the network), and a *primitive reduction relation* $e_1, \sigma_1 \rightsquigarrow e_2, \sigma_2; e_{f_1}, \cdots, e_{f_n}$ that relates an expression e_1 and a state σ_1 to an expression e_2, a state σ_2, and a (possibly empty) list $e_{f_1}, \ldots, e_{f_n}$ of expressions, corresponding to the threads forked by the reduction. A value denotes an expression that has reached its final form and will no longer reduce. We write $K[e]$ for the result of replacing the hole in evaluation context K with e.

The global state of the system is a *configuration* $c = (\text{tp}, \sigma)$, where the *thread pool* tp is a finite mapping from *locales*[5] to expressions, each corresponding to an execution thread. We will write $\text{tp}(\zeta)$ for the expression whose locale is ζ in tp and use $\text{tp}[\zeta \mapsto e]$ for the corresponding update (which adds a new thread if $\zeta \notin \text{dom}(\text{tp})$). We will write $\{\zeta \mapsto e\}$ for the singleton thread-pool consisting of a single thread e. For a language with shared-memory concurrency a locale would simply be a thread identifier. For a distributed language, a natural definition would be a pair (n, tid) of the name n of the node and the thread identifier tid of the thread in that node. Having explicit locales as part of the language definition will be beneficial when expressing, *e.g.*, thread-level properties such as fair scheduling.

---

[4]The compromise that Transfinite Iris makes is that it no longer validates some of the basic reasoning principles regarding step-indexing, *i.e.*, rules regarding interaction of the later modality with other connectives of the logic. It is partly due to this limitation that we opt to base our work on ordinary Iris at the cost of the relative image-finiteness side-condition.
[5]The name is inspired by the Chapel programming language, where it denotes the abstract place where programs execute.

The primitive reduction relation is lifted to an LTS by a relation between configurations labeled by the step-taking locale as follows:

$$\frac{(e_1, \sigma_1) \rightsquigarrow (e_2, \sigma_2; e_{f_1}, \cdots, e_{f_n}) \qquad fr(\zeta, \mathsf{tp}) = \zeta_1, \zeta_2, \ldots}{(\mathsf{tp}[\zeta \mapsto K[e_1]], \sigma_1) \xrightarrow{\zeta} (\mathsf{tp}[\zeta \mapsto K[e_2], \zeta_1 \mapsto e_{f_1}, \ldots, \zeta_n \mapsto e_{f_n}], \sigma_2)}$$

where $fr(\zeta, \mathsf{tp})$ is an infinite sequence of fresh locales (not in $\mathrm{dom}(\mathsf{tp})$) derived from ζ, e.g., in case of distributed systems it would consist of fresh thread identifiers on the same node as ζ.

The Trillium Program Logic. The goal of the Trillium program logic is to establish an intensional refinement $c \precsim_\xi \mathsf{m}$ between singleton traces consisting of the initial program configuration c and a model state m. Proving this refinement is a matter of proving that ξ *always holds*, throughout the execution of the program starting in c, alongside a corresponding traversal of the model starting from m. This can be achieved by ensuring that the program execution makes *progress* in tandem with the model, in addition to the relation ξ being *preserved* throughout any such execution.

Conventional Iris-style weakest precondition predicates $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ guarantee *postcondition validity* of terminated programs, *progress* of program executions, and *preservation* of the invariants whose names are in the *mask* $\mathcal{E}$. This can be seen explicitly in its definition; a guarded fixpoint of the following equation [Jung et al. 2018] (which, unlike in Trillium, does not consider locales):

$$
\begin{aligned}
\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq\ & \left(e \in Val * \Phi(e)\right) \vee & \textit{(post condition)} \\
& \left(e \notin Val * \forall \sigma.\, S(\sigma) \mathbin{-\!\!*} {}^{\mathcal{E}}\!\!\Rrightarrow^{\emptyset} reducible(e, \sigma)\ *\right. & \textit{(progress)} \\
& \quad \triangleright \forall e', \sigma', e_{f_1}, \cdots, e_{f_n}.\, (e, \sigma) \rightsquigarrow (e', \sigma'; e_{f_1}, \ldots, e_{f_n}) \mathbin{-\!\!*} \\
& \quad {}^{\emptyset}\!\!\Rrightarrow^{\mathcal{E}} S(\sigma') * \mathsf{wp}_{\mathcal{E}}\, e'\, \{\Phi\} * \underset{1 \le i \le n}{\text{\Large $\ast$}} \mathsf{wp}_{\top}\, e_{f_i}\, \{\Psi\} & \textit{(preservation)}
\end{aligned}
$$

The definition is by case distinction: either e is a value, in which case the postcondition should hold, or e is not a value, in which case there are two requirements (ignoring ${}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2}$ and $S(\sigma)$ for now). First, for the current state (captured by $S(\sigma)$), the program should be *reducible*, i.e., it *can* make progress. Second, for any program e' and forked threads e_{f_i} that e might reduce to, the weakest precondition must hold as well (with some post condition Ψ for the forked threads). The *later modality* $\triangleright$ guarantees that the fixpoint is well-defined (the recursive occurrence is guarded). The *update modality* ${}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P$ enables a form of rely-guarantee reasoning regarding invariants: to establish P the prover can access the invariants in $\mathcal{E}_1$ but they must also establish all the invariants in $\mathcal{E}_2$ alongside proving P. Hence, the definition of the weakest precondition preserves invariants in $\mathcal{E}$ by giving access to all of them (by going from $\mathcal{E}$ to $\emptyset$) but requires them to be preserved by asking them to be closed immediately after each program step (by going back from $\emptyset$ to $\mathcal{E}$). The predicate $S : State \to iProp$ is the *state interpretation* predicate that reflects the state (e.g., the heap) of the program as resources in the logic and gives meaning to, e.g., the traditional separation logic connective $\ell \mapsto v$ for heap ownership. Note how the definition of weakest precondition enforces that the state interpretation, just like invariants, is preserved throughout program execution. We often write $\mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, Q\} \triangleq \mathsf{wp}_{\mathcal{E}}\, e\, \{\lambda v.\, Q\}$, and $\mathsf{wp}_{\mathcal{E}}\, e\, \{Q\} \triangleq \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, v = () * Q\}$.

Remark 3.1 (Invariants and ghost resources in Iris). That P is invariant in Iris is represented by the proposition $\boxed{P}^{\mathcal{N}}$ which is annotated with a *name* $\mathcal{N}$ that identifies it. In order to work with invariants formally in Iris, the update modality is annotated with two masks: ${}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2}$. We write $\Rrightarrow_{\mathcal{E}}$ when $\mathcal{E}_1 = \mathcal{E}_2 = \mathcal{E}$ and $\Rrightarrow$ when $\mathcal{E} = \top$, the set of all masks. The update modality allows us to update ghost resources as described by Iris's ghost resource theory

---

[Jung et al. 2018] and to access invariants. Intuitively, the proposition ${}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P$ holds if we can establish P and all invariants in $\mathcal{E}_2$ through ghost updates, without violating the environment's resources (a "frame preserving update"), and all the invariants in $\mathcal{E}_1$. For weakest preconditions (in both the Iris and Trillium program logic) we can manipulate resources and invariants throughout the proof because weakest preconditions are closed under the update modality: $\Rrightarrow_{\mathcal{E}} \mathsf{wp}\, e\, \{\Phi\} \dashv\vdash \mathsf{wp}\, e\, \{\Phi\} \dashv\vdash \mathsf{wp}\, e\, \{x.\, \Rrightarrow_{\mathcal{E}} \Phi(x)\}$. The following rules allow us to create and access invariants and to manipulate the update modality. Note how accessing an invariant only makes its contents available one step later (under ▷); similarly we only need to prove it one step later to (re)establish the invariant. The INV-ACCESS rule—alongside providing the contents of the invariant—also tells us how we can close/reestablish the invariant.

$$
\begin{array}{c}
\text{INV-ALLOC} \\
\triangleright P \vdash \Rrightarrow_{\mathcal{E}} \boxed{P}
\end{array}
\qquad
\begin{array}{c}
\text{INV-ACCESS} \\
\mathcal{N} \in \mathcal{E} \\ \hline
\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \mathbin{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True})
\end{array}
\qquad
\begin{array}{c}
\text{UPD-MONO} \\
P \vdash Q \\ \hline
{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q
\end{array}
$$

$$
\begin{array}{c}
\text{UPD-MASK-TRANS} \\
{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2}\; {}^{\mathcal{E}_2}\!\!\Rrightarrow^{\mathcal{E}_3} Q \\ \hline
{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_3} Q
\end{array}
\qquad\qquad
\begin{array}{c}
\text{UPD-MASK-WEAKEN} \\
\mathcal{E}_2 \subseteq \mathcal{E}_1 \qquad {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_3} Q \\ \hline
{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2}\; {}^{\mathcal{E}_2}\!\!\Rrightarrow^{\mathcal{E}_3} Q
\end{array}
$$

These rules, along with the definition of weakest preconditions above allow us to prove a rule for accessing invariants during an "atomic" step of computation, *i.e.*, a program that reduces to a value in a single step of computation.

$$
\begin{array}{c}
\text{WP-ATOMIC} \\
\mathcal{N} \in \mathcal{E} \qquad \boxed{P}^{\mathcal{N}} \qquad \triangleright P \mathbin{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\mathcal{N}\}}\, e\, \{x.\, \triangleright P * \Phi(v)\} \qquad \mathrm{atomic}(e) \\ \hline
\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}
\end{array}
$$

Note how WP-ATOMIC might appear to *consume* the invariant. This is not an issue, though, as invariants are *persistent* and hence freely duplicable and shareable [Jung et al. 2018].

In the rest of this paper we will make heavy use of one kind of ghost resource, which we explain in terms of abstract predicates. Given a set A we define two predicates $\bullet_\gamma(a)$ and $\circ_\gamma(a)$, for any $a \in A$, respectively called the full part and the fragment of the resource; γ is the name of the resource instance used for disambiguation. These predicates are defined in terms of Iris ghost resources internally and satisfy the following rules, which essentially say that the full part and the fragment must always agree:

$$
\begin{array}{c}
\text{AUTH-AGREE} \\
\bullet_\gamma(a) * \circ_\gamma(b) \vdash a = b
\end{array}
\qquad\qquad
\begin{array}{c}
\text{AUTH-ALLOC} \\
\vdash \Rrightarrow_{\mathcal{E}} \exists \gamma.\, \bullet_\gamma(a) * \circ_\gamma(a)
\end{array}
\qquad\qquad
\begin{array}{c}
\text{AUTH-UPDATE} \\
\bullet_\gamma(a) * \circ_\gamma(a) \vdash \Rrightarrow_{\mathcal{E}} \bullet_\gamma(b) * \circ_\gamma(b)
\end{array}
$$

We refer to Jung et al. [2018] for a more thorough treatment of how invariants, the update modality, the later modality, and ghost state is constructed in Iris.

To define the Trillium program logic, we enrich the Iris-style weakest precondition to consider (1) program traces, (2) locales, and (3) a lock-step relation between the program trace and the model trace. Formally, this is defined as a guarded fixpoint of the equation below:

$$
\begin{aligned}
\mathsf{wp}_{\mathcal{E}}^{\zeta}\, e\, \{\Phi\} \triangleq {}& \\
& \bigl(e \in Val * \forall \tau, \kappa.\, S(\tau, \kappa) \mathbin{-\!\!*} \Rrightarrow_{\mathcal{E}} S(\tau, \kappa) * \Phi(e)\bigr) \vee \\
& \bigl(e \notin Val * \forall \tau, \kappa, \sigma, K, \mathrm{tp}.\, \mathit{last}(\tau) = (\mathrm{tp}[\zeta \mapsto K[e]], \sigma) * S(\tau, \kappa) \mathbin{-\!\!*} {}^{\mathcal{E}}\!\!\Rrightarrow^{\emptyset} \\
& \quad\ \mathit{reducible}(e, \sigma) * \triangleright \forall e', \sigma', e_{f_1}, \cdots, e_{f_n}.\, (e, \sigma) \rightsquigarrow (e', \sigma'; e_{f_1}, \cdots, e_{f_n}) \mathbin{-\!\!*} {}^{\emptyset}\!\!\Rrightarrow^{\mathcal{E}} \\
& \quad\ \exists \mathrm{m} \in \mathcal{M}, \rho.\, \mathit{last}(\kappa) \xrightarrow{\rho} \mathrm{m} *
\end{aligned}
$$

$$S(\tau \xrightarrow{\zeta} (\text{tp}[\zeta \mapsto K[e'], fr(\zeta, \text{tp})_1 \mapsto e_{f_1}, \dots, fr(\zeta, \text{tp})_n \mapsto e_{f_n}], \sigma'), \kappa \xrightarrow{\rho} \text{m}) *$$
$$\text{wp}_{\mathcal{E}}^{\zeta} e' \{\Phi\} * \underset{1 \le i \le n}{\text{\Large$*$}} \text{wp}_{\mathcal{E}}^{fr(\zeta, \text{tp})_i} e_{f_i} \{\Psi\})$$

Instead of just program configurations, the definition now considers all program traces τ where the expression e is about to make a step at the locale ζ, under an evaluation context K; and for all steps that e may take, there must exist a model state $\text{m} \in \mathcal{M}$ that the last state of the model trace κ can step to. Moreover, instead of a state interpretation, it tracks a *trace interpretation* S of the program and model traces, which will allow us not only to interpret the state of the language and the current model state as resources. Note also that in the case where e is a value, in order to establish the postcondition, we can also access the trace interpretation. This change is required for the soundness of some of our proof rules which apply even when the program is a value but whose correctness relies on the resources in the trace interpretation—we will discuss this later on.

That the definition of the Trillium weakest precondition has the intended meaning is the content of the following general adequacy theorem.

THEOREM 3.2 (ADEQUACY). *Let e be a program, σ a program state, ζ the locale of e in an otherwise empty thread pool, and Φ an Iris predicate on values. Let $\text{m} \in \mathcal{M}$ be a model state and ξ a relative image-finite relation on finite traces of the program and the model. Let $c = (\{\zeta \mapsto e\}, \sigma)$ be the initial configuration of the program. If $\xi(c, \text{m})$ holds for the initial singleton traces, and furthermore we have*

$$\models_\top S(c, \text{m}) * wp_\top^\zeta e \{\Phi\} * AlwaysHolds(\xi, c, \text{m})$$

then $c \preceq_\xi \text{m}$ holds in the meta-logic.

In particular, we must show that the trace interpretation holds for the initial singleton traces, that the weakest precondition holds, and that the predicate $AlwaysHolds(\xi, c, \text{m})$ holds. The predicate $AlwaysHolds(\xi, c, \text{m})$ is an Iris predicate that states that ξ (which is a relation on finite *traces*, not mentioning resources of the logic) does in fact always hold, *assuming* that all invariants hold, that postconditions hold (for any thread that has terminated up to that point in the trace), that the trace interpretation holds, and that the program never gets stuck. We refer to our accompanying Coq development for the precise definition of the *AlwaysHolds* predicate. In practice, designing program logics on top of Trillium is thus a matter of carefully picking a trace interpretation that admits user-friendly reasoning principles while allowing the user to conclude strong ξ relations.[6]

The adequacy theorem of Trillium is much stronger than that of the ordinary Iris program logic. Recall that the usual adequacy theorem only establishes postcondition validity, progress, and invariant preservation. Trillium's adequacy theorem, on the other hand, establishes intensional refinement for any relation ξ which in turn can include all three aforementioned guarantees. In addition, as per Lemma 2.4, it allows us to show that infinite traces of execution of the program also have corresponding infinite model traces. Consequently, the adequacy theorem of Trillium is much more complicated to establish. The proof is in two stages. We first show that the weakest precondition implies a finite approximation of intensional refinement stated in terms of guarded recursion of Iris's base logic. We then show that the aforementioned guarded recursive definition implies intensional refinement—this is where we exploit relative image-finiteness. We refer to the accompanying Coq development for the details of the proof.

---

[6]In fact, Trillium is conservative in that one can always pick a trivial model LTS, 🦆, and ξ yielding no further intricacies in the program logic, and the regular safety guarantees of Iris.

4 THE FAIRIS LOGIC

In this section we present the Fairis logic; an instantiation of Trillium for proving liveness properties under fairness assumptions, such as fair termination, of programs written in an OCaml-like programming language. Most non-trivial concurrent programs only enjoy liveness properties, including termination, under fair thread scheduling. To see this, consider the program in Figure 3. In this example, the function yn_start creates a new reference b initialized at 0, and starts two threads: yes swaps the value of b from **true** to **false** in a loop k times, and the no swaps it from **false** to **true** as many times. For thread yes to make progress, it has to wait for no to set the value of b to **true**, and *vice versa*. Clearly, this program would not terminate with an unfair scheduler which only gives execution time to, say, yes. But this program *does* terminate in practice, because operating systems' schedulers and processors do behave fairly. We discuss the details of the proof of termination of the example in Figure 3 in §4.2.1.

Our approach to taking fairness into account when reasoning about liveness properties is to incorporate it into the model LTS as follows. First, we use the labels of the model LTS $\mathcal{M}$, which we will henceforth call *roles* in this section, in order to be able to express fairness at the level of the model. Intuitively, a role is the abstraction of the concept of a thread at the model level. Bear in mind, though, that, as we will discuss, threads and roles are not always in one-to-one correspondence. To this end, we introduce a construction Fuel($\mathcal{M}$) which augments the model $\mathcal{M}$ to keep track of the correspondence between threads and roles in order to provide finite stuttering. This construction also adapts the relation ξ and lifts it to a relation Fuel(ξ) between traces of the program and traces of Fuel($\mathcal{M}$). For brevity, we will write ξ_{fuel} instead of Fuel(ξ).

In order to show that the program enjoys a certain liveness property P, *i.e.*, that all *fair* execution traces τ of the program satisfy $P(\tau)$, we carry out the proof in the following steps:

(1) We choose a model $\mathcal{M}$ and a relation ξ.
(2) We prove a specification for our program using the Fairis program logic whose adequacy theorem ensures that all program traces (including unfair executions) refine some trace of the instrumented model Fuel($\mathcal{M}$) for the relation ξ_{fuel}.
(3) We have proved (once and for all) that given two traces τ and τ', if $\hat{\xi}_{fuel}(\tau, \tau')$, and τ is fair, then so is τ'. (Intuitively, this holds because we have constructed Fuel($\mathcal{M}$) and ξ_{fuel} such that they only allow finite stuttering in each thread.)
(4) We prove that all fair traces of $\mathcal{M}$ satisfy the desired property P.
(5) We prove that P is preserved by ξ_{fuel}, in that $\hat{\xi}_{fuel}(\tau, \tau')$ and $P(\tau')$ implies $P(\tau)$.
(6) We prove that, if for a trace τ in Fuel($\mathcal{M}$) we have that the corresponding trace τ' satisfies P, then so does τ, this roughly corresponds to the fact that P is invariant under finite stuttering.
(7) From all of the above, we conclude $P(\tau)$ for any fair trace τ of the program.

This may seem like a lot of steps, but (7) is automatic, and given a fixed property P, steps (5) and (6) can be proved independently of the program and of the model $\mathcal{M}$. Step (6) is trivial for the case of termination and properties of the always-eventually form as these are preserved under finite stuttering. Finally, in case of termination, we have formulated a simple criterion for step (4) which is local in the sense that one does not need to consider traces but rather need to inspect individual transitions in $\mathcal{M}$. See our accompanying appendix and Coq formalization for this local criterion.

What the user obtains from the adequacy theorem of Fairis is $\preccurlyeq_{\xi_{fuel}}$ between the program and the model LTS Fuel($\mathcal{M}$); ξ_{fuel} is relative image-finite by construction whenever ξ is. Crucially, all the liveness properties we are interested in—such as termination or "always-eventually" trace properties—are stable under finite stuttering. Hence, if all fair traces of $\mathcal{M}$ satisfy our property so do all fair traces of Fuel($\mathcal{M}$). Consequently, the user only has to prove that the property can be transported along the $\preccurlyeq_{\xi_{fuel}}$ intensional refinement relation from Fuel($\mathcal{M}$) to the program. In
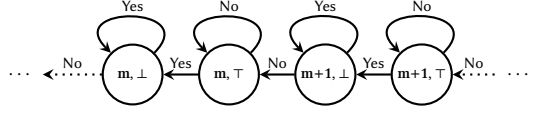
```
let rec yes b n =
  if cas b true false then n := !n-1;
  if !n > 0 then yes b n

let rec no b m =
  if cas b false true then m := !m-1;
  if !m > 0 then no b m

let yn_start k = let b = ref true in
  (yes b (ref k) || no b (ref k))
```

Fig. 3. The Yes and No threads.

We write $\top$ for **true** and $\bot$ for **false**.

Fig. 4. The model $\mathcal{M}_{\text{yes\_no}}$.

some cases, *e.g.*, §4.2.3, we may be interested in proving a liveness property under an assumption stronger than fairness, *i.e.*, that all fair traces that satisfy a certain property Q satisfy the desired liveness property P. In such cases the user of Fairis is responsible for proving that the property Q can be transported from the program to $\text{Fuel}(\mathcal{M})$ and from $\text{Fuel}(\mathcal{M})$ to $\mathcal{M}$—in case of the example in §4.2.3, we are interested in infinite fair traces; infinitude, like finiteness, trivially transports as it is preserved by intensional refinement and finite stuttering. The Fuel construction essentially augments the model LTS to add two pieces of information: (1) for each thread id ζ, the set of roles that are associated to ζ, and, (2) for each role ρ, the value of its so-called *fuel*, *i.e.*, a number which measures how long the program can still postpone taking a step in that role in ξ. This allows us to assign *obligations* to threads, *i.e.*, assign a set of *live* roles to a thread—a role is said to be live if it can still take steps. A thread that has a set R of roles assigned to it must eventually (restricted by fuel) take a step in each of those roles. This is enforced by mandating that to take a step of computation, a thread must have a non-empty set R of live roles, and that after the step all fuels in R are decreased, except if this step was matched by a non-stuttering step corresponding to a role $\rho \in R$ (one of its live roles) in the underlying LTS $\mathcal{M}$. In that case, the thread may increase the fuel of role ρ (but the fuel for all other roles in R must decrease) as long as it remains under a certain global cap which we will denote with F_{cap}.[7] Alternatively, a thread can *delegate* some of its obligations to a thread that it forks. The adequacy theorem of Fairis requires a proof in the program logic that enforces that threads may only stop if there are no live roles assigned to them anymore; this is encoded through threads' postconditions.

The Fairis adequacy theorem is a special case of the Trillium adequacy theorem. The theorem is shown below, and has differences from the Trillium adequacy theorem highlighted:

THEOREM 4.1 (FAIRIS-ADEQUACY). *Let e be a program, σ a program state, ζ the locale of e in an otherwise empty thread pool. Let $\mathsf{m} \in \mathcal{M}$ be a model state, and ξ a relative image-finite relation on program traces and model traces. Let $c = (\{\zeta \mapsto e\}, \sigma)$ be the initial configuration of the program, and $\tilde{\mathsf{m}}$ be the initial state of $\text{Fuel}(\mathcal{M})$ corresponding to m, i.e., m together with ζ assigned to a map F_{init} of all roles with F_{cap} fuel. If $\xi(c, \mathsf{m})$ and*

$$\Rrightarrow_{\top} (\circ_{\gamma_{\mathcal{M}}}(\mathsf{m}) * \zeta \mapsto \mathsf{F}_{init} * \underset{\ell;v\in\sigma}{\text{\Large{$*$}}} \ell \mapsto v) \twoheadrightarrow$$

$$S(c, \tilde{\mathsf{m}}) * wp_{\top}^{\zeta} e \{w. \zeta \mapsto \emptyset\} * AlwaysHolds(\xi, c, \tilde{\mathsf{m}})$$

then $c \precsim_{\xi_{fuel}} \tilde{\mathsf{m}}$ holds in the meta-logic.

Here, $\circ_{\gamma_{\mathcal{M}}}(\mathsf{m})$ and $\text{tid} \mapsto \mathsf{F}_{init}$ are initial resources used in the Fairis program logic rules. In particular, $\circ_{\gamma_{\mathcal{M}}}(\mathsf{m})$ is an exclusive resource that tracks the current state of the user model, while

---

[7]This global cap, fixed for each refinement proof, ensures relative image-finiteness (Definition 2.6) of $\text{Fuel}(\mathcal{M})$.

$\zeta \mapsto \mathit{fs}$ associates the locale ζ with a partial map $\mathit{fs} : \mathrm{Role}(\mathcal{M}) \rightharpoonup \mathbb{N}$ that associates model roles with their fuels. The authoritative part $\bullet_{\gamma_{\mathcal{M}}}(m)$ of the resource is kept as part of the trace interpretation; the name $\gamma_{\mathcal{M}}$ is a globally fixed name for the entire proof created as part of the proof of the adequacy theorem. The resource $\circ_{\gamma_{\mathcal{M}}}(m)$ is available to the user of the Fairis program logic to be able to relate the state of the program to the state of the model (which always agrees with the one tracked in the trace interpretation as per resource rules), usually in an invariant. This relation in the logic is then what the user will use to establish the ξ relation as required—we will see this in examples below. Initially, all roles are assigned to the single thread in the thread-pool with fuels all being F_{cap}. The last piece of resource the user acquires from the adequacy theorem is the full ownership of the entire initial state (heap) σ in that we obtain points-to propositions for all locations in σ. The adequacy theorem, and the rule for forking threads as we will see later, enforce that upon termination (in the postcondition) the thread must own no live roles. Finally, note that even though the adequacy theorem establishes a $\precsim_{\xi_{fuel}}$ relation, the user only needs to prove *AlwaysHolds* for ξ and not for ξ_{fuel}.

In the rest of this section we cover how we apply Fairis to prove various liveness properties. First, we present the key ideas of the Fairis program logic, along with its reasoning rules, used for proving the weakest precondition of programs §4.1. We then give a tour of how the Fairis program logic can be used to prove liveness properties of programs §4.2, including fair termination of the yes-no example discussed earlier. Finally, we outline some of the technical details involved in obtaining the Fairis adequacy theorem §4.3.

The examples that we present in this section are simple examples designed to demonstrate viability of the Trillium approach to liveness properties under fair scheduling assumptions. As such, in all these examples the LTS model used includes the entire core functionality of the program; only the administrative program steps, *e.g.*, beta-reduction, are not included which are taken into account via stuttering enabled by the fuel construction. Hence, in that sense, the LTS models of these examples are not very abstract. For more involved examples, we expect that the level of abstraction of the LTS model compared to the program should be similar to that of Aneris examples; c.f. the Paxos example of §5 where the model abstracts away a substantial amount of implementation details. However, to substantiate this claim one needs a proper experimental evaluation, using a wide range of more advanced examples, which is beyond the scope of the current work, and thus we leave it for future work.

4.1 The Fairis Program Logic

The Fairis program logic combines the reasoning principles of conventional Iris program logics with reasoning principles involving fuel and model resources. To properly compartmentalize these orthogonal reasoning principles, the program logic employs two layers of reasoning: an outer (model) layer and an inner (program) layer.

The intuition is that at every program step, the program will either do a *fuel step*, corresponding to stuttering, or a *model step*, corresponding to a step in the underlying model. Both of these are handled by the outer model layer. Regardless of which kind of step the program takes, the inner layer will then be used to reason about the actual program step. The rules of the inner program logic only concern the program on its own and hence very closely resemble those of the ordinary program logic of Iris.

The inner program logic layer is expressed in terms of a different weakest precondition $\mathsf{wp}_{\mathcal{E}}\, e\, \langle \Phi \rangle$ (note the angle brackets instead of braces for the postcondition). This weakest precondition is a (heavily) simplified version of Trillium's weakest precondition where the state interpretation (it indeed only mentions the state and not the entire trace) only concerns the heap of the program. In

Fuel & model rules (outer program logic)

WP-STEP-FUEL
$$\frac{\zeta \Mapsto fs^{++} \qquad fs \neq \emptyset}{\mathsf{wp}_{\mathcal{E}}\, e\, \langle e'. \zeta \Mapsto fs \ast \mathsf{wp}^{\zeta}_{\mathcal{E}}\, e'\, \{\Phi\}\rangle}{\mathsf{wp}^{\zeta}_{\mathcal{E}}\, e\, \{\Phi\}}$$

WP-ROLE-DEALLOC
$$\frac{\circ_{\gamma_M}(m) \qquad m \not\xrightarrow{} \_ \qquad \zeta \Mapsto \{\rho := \_\} \uplus fs}{(\circ_{\gamma_M}(m) \ast \zeta \Mapsto fs) \mathbin{-\!\!*} \mathsf{wp}^{\zeta}_{\mathcal{E}}\, e\, \{\Phi\}}{\mathsf{wp}^{\zeta}_{\mathcal{E}}\, e\, \{\Phi\}}$$

WP-STEP-MODEL
$$\frac{\circ_{\gamma_M}(m) \qquad m \xrightarrow{\rho} m' \qquad \zeta \Mapsto \{\rho := \_\} \uplus (fs^{++})}{\mathsf{wp}_{\mathcal{E}}\, e\, \langle e'. (\circ_{\gamma_M}(m') \ast \zeta \Mapsto \{\rho := \mathsf{F}_{cap}\} \uplus fs) \mathbin{-\!\!*} \mathsf{wp}^{\zeta}_{\mathcal{E}}\, e'\, \{\Phi\}\rangle}{\mathsf{wp}^{\zeta}_{\mathcal{E}}\, e\, \{\Phi\}}$$

WP-ROLE-FORK
$$\frac{\zeta \Mapsto (fs_1 \uplus fs_2)^{++} \qquad fs_1 \uplus fs_2 \neq \emptyset \qquad \forall \zeta'. \zeta' \Mapsto fs_2 \mathbin{-\!\!*} \mathsf{wp}^{\zeta'}_{\mathcal{E}}\, e\, \{\zeta' \Mapsto \emptyset\}}{\mathsf{wp}^{\zeta}_{\mathcal{E}}\, \mathbf{fork}\,(e)\, \{\zeta \Mapsto fs_1\}}$$

Program rules (inner program logic; an excerpt)

WP-ALLOC
$$\mathsf{wp}_{\mathcal{E}}\, \mathbf{ref}\, v\, \langle w.\exists \ell.\, w = \ell \ast \ell \mapsto v\rangle$$

WP-LOAD
$$\frac{\ell \mapsto v}{\mathsf{wp}_{\mathcal{E}}\, !\,\ell\, \langle w.\, w = v \ast \ell \mapsto v\rangle}$$

WP-STORE
$$\frac{\ell \mapsto v}{\mathsf{wp}_{\mathcal{E}}\, \ell \leftarrow w\, \langle \ell \mapsto w\rangle}$$

WP-CAS
$$\frac{\ell \mapsto v}{\mathsf{wp}_{\mathcal{E}}\, \mathbf{cas}\, \ell\, v_1\, v_2 \left\langle \begin{matrix} w.\, (v = v_1 \ast w = \mathbf{true} \ast \ell \mapsto v_2)\ \vee \\ (v \neq v_1 \ast w = \mathbf{false} \ast \ell \mapsto v) \end{matrix} \right\rangle}$$

WP-BETA
$$\mathsf{wp}_{\mathcal{E}}\, (\lambda x.\, e)\, v\, \langle e'.\, e' = e[v/x]\rangle$$

Fig. 5. The rules of the Fairis program logic

particular, this inner program logic on its own would be useless. It is only useful, and extremely so, because the outer program logic has already taken care of the details of the model including the fuel. Additionally, the inner weakest precondition strictly captures a *single* step of reduction as for further steps the outer program logic must intervene again to manage the model-side details. As a result, the postcondition of the inner weakest preconditions are predicates over arbitrary expressions as opposed to values, *i.e.*, a predicate $\Phi : Expr \to iProp$. Similar to the outer weakest precondition, we often write $\mathsf{wp}_{\mathcal{E}}\, e\, \langle e'.\, Q\rangle \triangleq \mathsf{wp}_{\mathcal{E}}\, e\, \langle \lambda e'.\, Q\rangle$, and $\mathsf{wp}_{\mathcal{E}}\, e\, \langle Q\rangle \triangleq \mathsf{wp}_{\mathcal{E}}\, e\, \langle e'.\, e' = () \ast Q\rangle$. Additionally, we often use w instead of e' for the return value binder, to denote that the returned expression is a value. The definition of the inner weakest precondition is given in the accompanying appendix.

An overview of the layers and their rules can be found in Figure 5. The rules of the inner program logic are entirely standard. We will thus not explain them in detail. The WP-STEP-FUEL rule captures that we can take a *fuel step* whenever the fuel of all the roles of the non-empty fuel map is non-zero ($\zeta \Mapsto fs^{++}$ and $fs \neq \emptyset$, where fs^{++} denotes the map fs where all fuels are incremented by 1). We must then prove the inner weakest precondition, where the postcondition captures that we re-obtain the fuel map ($\zeta \Mapsto fs$), where all fuel has been decremented, for the remaining proof obligation.

The WP-STEP-MODEL rule captures that we can take a *model step* whenever the underlying model can take a step over some role ρ; we have $\circ_{\gamma_M}(m)$ indicating the current model state, and $m \xrightarrow{\rho} m'$ indicating we can take a step. We must then show that the role ρ is associated with the current locale ζ, and that the fuel of all non-ρ roles are non-zero ($\zeta \Mapsto \{\rho := \_\} \uplus fs^{++}$). Just as in WP-STEP-FUEL we must show the inner weakest precondition, which in turn gives us back the model and fuel resources. The model resource is updated to the new model state, $\circ_{\gamma_M}(m')$, and the fuel map has the fuel of ρ replenished to the maximum fuel cap, F_{cap}, while all non-ρ fuels have been decremented, $\zeta \Mapsto \{\rho := \mathsf{F}_{cap}\} \uplus fs$. The conjunction of these rules effectively enforce, that for any role ρ for which we can only take finitely many steps in the thread that ρ is assigned to, we are forced to

take a step corresponding to ρ. Otherwise, we would run out of fuel for the role ρ and hence not be able to take anymore steps, be it a stuttering step, or a step of any role other than ρ.

The rules WP-ROLE-DEALLOC and WP-ROLE-FORK allow us to discharge our obligations (roles we are responsible for). The rule WP-ROLE-DEALLOC removes a role from the fuel map of the locale when all the obligations of that role are fulfilled, *i.e.*, when the role "terminates" and cannot take any more steps. The rule WP-ROLE-FORK, on the other hand, allows the thread to *delegate* some of its obligations (roles) by forking a thread and passing those roles to the newly forked threads. Note how the fuel is decremented for all roles, including those that are passed to the newly forked threads. This restriction is necessary for the soundness of the logic as otherwise one could indefinitely postpone taking a step in role by repeatedly forking new threads. Furthermore, note how the rule WP-ROLE-FORK requires the postcondition of the forked thread to have a live role when it terminates, effectively forcing it to discharge all its obligations before termination.

4.2 Examples

In this section we cover a suite of examples to demonstrate how Fairis can be used to prove fairness-dependent liveness properties. In particular, we first prove *fair termination* of the yes-no example (Figure 3) discussed earlier. We will then present an example in §4.2.2 where the model LTS is *not* image-finite but still the ξ relation is relative image-finite *cf.* Definition 2.6. Finally, we demonstrate how Fairis can be used to show liveness properties beyond termination in §4.2.3.

4.2.1 Fair Termination: Yes-No. We initially consider the application of Fairis for proving fair termination, which we formally state as follows over the initial configuration c of the program:

$$\mathsf{fairly\_terminating}(c) \triangleq \forall \tau. \mathsf{first}(\tau) = c \implies \mathsf{fair}(\tau) \implies \mathsf{finite}(\tau)$$

In particular, we prove fair termination for the yes-no program presented in Figure 3, using the associated model shown in Figure 4. Proving fair termination of the model is relatively straightforward. In our Coq formalization we apply the local criterion mentioned earlier; this criterion is explained in our accompanying appendix. Moreover, as discussed earlier, termination can be trivially transported from $\mathcal{M}$ to $\mathsf{Fuel}(\mathcal{M})$. Hence, to prove fair termination of the yes-no program, we just have to show that it is refined by the yes-no model using Fairis.

To do so, we apply the adequacy theorem of Fairis, using a trivial ξ relation, and an empty initial state σ for the program. This choice means that $AlwaysHolds(\mathsf{Fuel}(\xi), c, \tilde{\mathsf{m}})$ holds trivially for any c and $\tilde{\mathsf{m}}$. Since the yes-no model (Figure 4) is finitely branching, any ξ relation on it is trivially relative image-finite. What remains to prove is the weakest precondition of the yes-no program:

$$(\circ_{\gamma_{\mathcal{M}}}(\mathsf{m}) * \zeta \mapsto \{\mathsf{Yes} := \mathsf{F}_{cap}; \mathsf{No} := \mathsf{F}_{cap}\}) \mathrel{-\!\!*} \mathsf{wp}_{\top}^{\zeta} \, \mathsf{yn\_start} \, n \, \{\zeta \mapsto \emptyset\}$$

where m is the state $(n, \top)$ in Figure 4, and $0 < n$ is any *positive* natural number. As for applying the rules updating the underlying model, the only steps in the program where this happens are the two **cas** operations in the yes and no threads. These are precisely the horizontal steps in Figure 4 when the **cas** operation succeeds and the loop when the **cas** operation fails—as **cas** is *atomic*, we can access the invariant (see the rest of the proof argument) during a **cas** operation. All the other steps in the program are stuttering steps.

The proof starts by using WP-ALLOC to resolve the allocation of the flag reference, obtaining $l \mapsto$ **true**. As per conventional Iris methodology for concurrent programs, the proof employs an invariant to safely share the flag reference among the two threads. This invariant additionally ensures that the program behaves according to the model LTS by incorporating the $\circ_{\gamma_{\mathcal{M}}}$ part of the model resource. Recall that n corresponds to how many times the model must cycle between flipping the flag b back and forth before termination. In order to tie the state of the model to the

```
let rec decr_loop l =
  if !l > 0 then l := !l - 1 else #()

let nondet_start l =
  l := (nondet () + 1);; decr_loop l
```
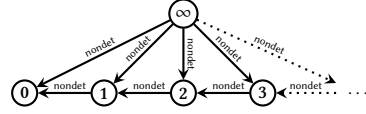
Fig. 6. The non-deterministic nat example.

Fig. 7. The non-deterministic nat model.

references managed by the two threads, we incorporate two additional resource predicates $\bullet_{\gamma_{yes}}$ and $\bullet_{\gamma_{no}}$ in the invariant, each tracking how many times each thread must still flip the flag b—the fragmental parts of these resources will be passed to the two threads. The invariant used to prove the yes-no program, $\mathsf{yesno\_inv}(\gamma_{yes}, \gamma_{no}, l)$, is as follows:

$$\mathsf{yesno\_inv}(\gamma_{yes}, \gamma_{no}, l) \triangleq \exists n, b. \circ_{\gamma_M}(n, b) * (n, b) \neq (0, \mathbf{false}) * l \mapsto b *$$
$$\left(b = \mathbf{true} * \bullet_{\gamma_{yes}}(n) * \bullet_{\gamma_{no}}(n)\right) \vee \left(b = \mathbf{false} * \bullet_{\gamma_{yes}}(n) * \bullet_{\gamma_{no}}(n-1)\right)$$

Note how in the case the flag is **false** the value of the no thread may lag behind because it has not yet had the chance to flip the flag in this "round" to be able to catch up. Furthermore, note how this invariant asserts that the state $(0, \mathbf{false})$ is not ever reached in the model when the program is refining the model. This is crucially the case because we only ever call yn_start with a strictly positive argument. It only remains to prove the following specifications of the two threads

$$\boxed{\mathsf{yesno\_inv}(\gamma_{yes}, \gamma_{no}, l)}^{N_{YN}} * \zeta \mapsto \{\mathsf{Yes} := 42\} * \circ_{\gamma_{yes}}(n) * 0 < n) \twoheadrightarrow \mathsf{wp}_\top^\zeta \mathsf{yes} \, l \, n \, \{\zeta \mapsto \emptyset\}$$
$$\boxed{\mathsf{yesno\_inv}(\gamma_{yes}, \gamma_{no}, l)}^{N_{YN}} * \zeta \mapsto \{\mathsf{No} := 42\} * \circ_{\gamma_{no}}(n) * 0 < n) \twoheadrightarrow \mathsf{wp}_\top^\zeta \mathsf{no} \, l \, n \, \{\zeta \mapsto \emptyset\}$$

Note that the fuel of 42 is picked somewhat arbitrarily; it only needs to be below F_{cap} (which we can arbitrarily pick) and large enough for the proof to go through.

We use these specifications along with wp-role-fork to delegate the role obligations to the two threads, which completes the proof of the main thread which can (and must) now terminate as it has no roles associated to it anymore. We use wp-step-fuel to resolve all the stuttering steps in between model steps. The crux of the rest of the proof is the **cas** operations, where we update the model using the rule wp-step-model, regardless of it succeeding or not. In particular, we open the invariant around the operation using the rule inv-access, and then resolve the operation with wp-cas. Based on whether the **cas** operation succeeds, we either take the decrementing or the looping step in the model. In either case, the invariant is preserved. If we succeed, we update our model resource $\circ_{\gamma_M}(n)$ accordingly to preserve the invariant. If we fail, we simply do nothing and loop. In both cases, a model step has taken place in the underlying logic and hence the fuel for thread's role can be replenished. The program loops until we eventually hit 0, in which case we can use wp-role-dealloc to discharge the role obligation, as the role can no longer step in the model, thus fulfilling the postcondition.

4.2.2 Sound Infinite Branching: Non-deterministic Nat.
In §1 we remarked that the model should be relatively image-finite for the Trillium (and thereby Fairis) adequacy theorem to be sound. In this section we consider a model which is infinitely branching, while still being relative image-finite, by virtue of the user picked relation ξ.

The program we consider can be seen in Figure 6. This program works on a pre-allocated reference ℓ as input. It then assigns a non-zero non-deterministically chosen natural number n to the reference, which it decrements until it hits zero, after which point it terminates. A specification for the non-deterministic number operation is simply wp_ε nondet $\langle w.\exists n. w = n\rangle$. The model of the program can be seen in Figure 7. It starts in an initial state ∞, from which it can go to any natural

```
let rec incr_loop l n =
  if cas l n (n+1) then incr_loop l (n+2) else incr_loop l n

let eo_start l =
  let x := !l in (incr_loop l x || incr_loop l (x+1))
```
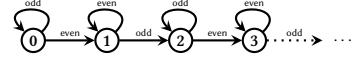
Fig. 8. The even-odd example.

Fig. 9. The even-odd model.

number n. While the model is terminating (all traces are finite), it is also infinitely branching to accommodate for the fact that the program may non-deterministically step to any natural number.

In order to prove relative image-finiteness we apply the Fairis adequacy theorem with the following user relation:

$$\xi_{\mathsf{nondet}}^{\ell}(\tau, \kappa) \triangleq (last(\kappa) = \infty \land \mathsf{heap}(last(\tau))(\ell) = -1) \lor (\exists n.\, last(\kappa) = n \land \mathsf{heap}(last(\tau))(\ell) = n)$$

That is, the state of the model always correspond to the number stored in the location, once it has been initialized—before that the location stores -1. This relation ensure relative image-finiteness, as for every program step, there is only *one* valid model step. Intuitively, the model side number we step to is uniquely (and hence finitely) determined by the program.

To be able to prove $AlwaysHolds(\xi_{\mathsf{nondet}}(\ell), c, \mathsf{m})$ we allocate a resource predicate, starting in the initial model state $\bullet_\gamma(\infty)$ and $\circ_\gamma(\infty)$, and use the former establish the following invariant at the very beginning of our verification:

$$\mathsf{nondet\_inv}(\gamma, \ell) \triangleq \exists cn.\, \circ_{\gamma_M}(cn) * \bullet_\gamma(cn) * (cn = \infty * \ell \mapsto -1) \lor (\exists n.\ cn = n * \ell \mapsto n)$$

This invariant enforces that the location points to a natural number corresponding to the model. Additionally, whenever the model is in the initial ∞ state, the location stores -1. Note that the relation ξ follows directly from the invariant.

To prove fair termination of the program, it only remains to prove the following specification:

$$\left(\boxed{\mathsf{nondet\_inv}(\gamma, \ell)} \right)^{N_{\mathsf{nondet}}} * \circ_\gamma(\infty) * \zeta \Mapsto \{\mathsf{nondet} := \mathsf{F}_{cap}\}) \twoheadrightarrow \mathsf{wp}_{\top}^{\zeta}\ \mathsf{nondet\_start}\ \ell\ \{\zeta \Mapsto \emptyset\}$$

It is straightforward to prove that the invariant holds throughout the program, and that it implies the user defined relation. The only non-stuttering step in the proof is the assignment step in the then branch of `decr_loop` which allows us to replenish the fuel in every loop iteration. An interesting aspect of this proof is that we use an invariant in the proof despite the fact that program is not concurrent. In this case, the invariant is not used to facilitate sharing between threads but rather to enforce *an invariant* of the program, *i.e.*, that the value of the reference must always, after initialization, correspond to what is determined in the model LTS.

4.2.3 Liveness Properties Beyond Termination: Even-Odd. In this section we demonstrate how Fairis can be used to prove liveness properties beyond termination.

Consider the program shown in Figure 8. The program works on a location ℓ assumed to store 0 initially. It forks off two threads that each indefinitely increment the location whenever it is even or odd, respectively. The property that we prove is that the counter ℓ visits *all* natural numbers *in order*. This property only makes sense for maximal traces, *i.e.*, traces that are either infinite or can take no further step—as we will discuss, all maximal traces of this program are infinite. Formally, the property we prove is the following:

$$\mathsf{evenodd\_prop}(\ell, c) \triangleq \forall \tau.\, \mathsf{first}(\tau) = c \Rightarrow \mathsf{fair}(\tau) \Rightarrow \mathsf{maximal}(\tau) \Rightarrow$$

$$(\forall i.\, \exists n.\, \mathsf{heap}(\tau(n))(\ell) = i) \land \qquad\qquad \textit{(all numbers visited)}$$

$$(\forall n.\, \exists i.\, \mathsf{heap}(\tau(n))(\ell) = i \land \exists j.\, i \le j \land \tau(n+1)(\ell) = j) \qquad \textit{(monotonicity)}$$

We prove the property above by relating the program in Figure 8 with the model LTS Figure 9. To this end, we prove the following property about the model Figure 9.

$$\text{evenodd\_prop\_mdl}(\ell, m) \triangleq \forall \kappa. \, \text{first}(\kappa) = m \Rightarrow \text{fair}(\kappa) \Rightarrow \text{infinite}(\kappa) \Rightarrow$$

$$(\forall i. \, \exists n. \, \tau(n) = i) \wedge \qquad \qquad \textit{(all number visited)}$$

$$(\forall n. \, \exists i. \, \tau(n) = i \wedge \exists j. \, i \leq j \wedge \tau(n+1) = j) \qquad \textit{(monotonicity)}$$

First, the property $\text{evenodd\_prop\_mdl}(\ell, m)$ is stable under finite stuttering—monotonicity is expressed using $\leq$ which is preserved by stuttering. Second, notice the discrepancy between this property and $\text{evenodd\_prop}(\ell, c)$ where the latter only assumes maximality of the trace while the former requires infinitude. This property is easy to prove under fairness assumptions. A fair infinite trace cannot have a tail that just consists with one of the loops. Hence, it must visit all numbers. Monotonicity follows rather trivially.

To finish the proof, we need to establish that (1) any maximal execution trace is infinite, and (2) that the liveness property can be transported from the fuel-instrumented model to the execution trace. To this end, we will pick ξ relations as follows:

$$\xi^\ell_{\text{even\_odd}}(\tau, \kappa) \triangleq \xi_{\text{steps}}(\tau) \wedge \xi^\ell_{\text{match}}(\tau, \kappa)$$

$$\xi_{\text{steps}}(\tau) \triangleq \exists \zeta, c'. \, last(\tau) \xrightarrow{\zeta} c'$$

$$\xi^\ell_{\text{match}}(\tau, \kappa) \triangleq \forall n < length(\tau). \, \tau(n)(\ell) = \kappa(n)$$

Note that, again, because the model in Figure 9 is finitely branching, the relation $\xi_{\text{even\_odd}}$ is trivially relative image-finite. Moreover, if we have $\tau \hat{\preceq}_{\xi^\ell_{\text{even\_odd}}} \kappa$, given the $\xi_{\text{steps}}(\tau)$ part of $\xi^\ell_{\text{even\_odd}}$ together with maximality of τ, we can conclude that τ must be infinite. This established by (1) above. The other part of the relation we pick, ξ^ℓ_{match}, simply expresses that the value stored in reference ℓ is always the same as the number in the model trace. In order to prove $\textit{AlwaysHolds}(\xi^\ell_{\text{even\_odd}}, c, m)$ we must show that ξ^ℓ_{steps} is preserved throughout program execution. This, however, is a simple consequence of post-condition validity and progress properties that we can assume as we prove $\textit{AlwaysHolds}$. On the one hand, we know that the program cannot be stuck, that is each thread is either a value or it can take further steps. On the other hand, we cannot have that all threads are values, as if they were, then their postconditions would all hold. This would allow us to conclude that we have reached a state on the model side that has no live roles, which is in contradiction with the definition of the model as given in Figure 9. Now, for establishing the other part of $\textit{AlwaysHolds}(\xi^\ell_{\text{even\_odd}}, c, m)$, we follow an approach similar to the two preceding examples. We use resource predicates $\bullet_{\gamma_e}(n), \circ_{\gamma_e}(n), \bullet_{\gamma_o}(n)$, and $\circ_{\gamma_o}(n)$, and we will use the full parts in the invariant below while the fragment parts are passed to the two threads. The invariant below immediately allows us to establish that ξ^ℓ_{match} is preserved throughout program's execution.

$$EO\_inv(\gamma_e, \gamma_o, \ell) \triangleq$$

$$\exists n. \, \circ_{\gamma_M}(n) * \ell \mapsto n * \big(iseven(n) * \bullet_{\gamma_e}(n) * \bullet_{\gamma_o}(n+1) \big) \vee \big(isodd(n) * \bullet_{\gamma_e}(n+1) * \bullet_{\gamma_o}(n) \big)$$

This invariant combines the ideas of the invariants of the preceding two examples. In particular, it lets us modularly prove that the number stored in ℓ corresponds to the model state. Note how in each case, whether n is even or odd, the number tracked by the odd and even threads, respectively, are out of sync by being one ahead of n. This very closely reflects the behavior of the program; the thread that is out of sync is waiting for the in-sync thread to perform its increment.

Finally, we need to prove the weakest precondition of the program.

$$\big(\boxed{EO\_inv(\gamma_e, \gamma_o, \ell)}\big)^{\mathcal{N}_{eo}} * \zeta \mapsto \{even := F_{cap}; odd := F_{cap}\} * \circ_{\gamma_e}(0) * \circ_{\gamma_o}(1) \twoheadrightarrow wp^\zeta_\top \, eo\_start \, \ell \, \{\zeta \mapsto \emptyset\}$$

	$f \xrightarrow{\text{tau}_\zeta} f'$	$f \xrightarrow{\text{vis}_\zeta(\rho_t)} f'$
ST	$f.m = f'.m$	$f.m \xrightarrow{\rho} f'.m$
TH	$\zeta \in \text{rng}(\text{th})$	$\text{th}(\rho) = \zeta$
DOM	$\text{roles}(f') \subseteq \text{roles}(f)$	$\text{roles}(f') \setminus \text{roles}(f) \subseteq \text{lroles}(f') \setminus \text{lroles}(f)$
DEC	$\forall \rho \in \text{dom}(f.\text{map}(\zeta)),\ \text{fm}'(\rho) < \text{fm}(\rho)$	$\forall \rho \in \text{dom}(f.\text{map}(\zeta)) \setminus \{\rho_t\},\ \text{fm}'(\rho) < \text{fm}(\rho)$
CH	$\forall \rho,\ \text{th}'(\rho) \neq \text{th}(\rho) \implies \text{fm}'(\rho) < \text{fm}(\rho)$	$\forall \rho \neq \rho_t,\ \text{th}'(\rho) \neq \text{th}(\rho) \implies \text{fm}'(\rho) < \text{fm}(\rho)$
NI	$\forall \rho \notin \text{dom}(f.\text{map}(\zeta)),\ \text{fm}'(\rho) \leq \text{fm}(\rho)$	$\forall \rho \notin \text{dom}(f.\text{map}(\zeta)),\ \text{fm}'(\rho) \leq \text{fm}(\rho)$
REF		$\forall \rho \in \text{roles}(f') \setminus \text{roles}(f) \cup \{\rho_t\},\ \text{fm}'(\rho) \leq \mathsf{F}_{cap}$

Fig. 10. Transitions between two states f and f' in $\text{Fuel}(\mathcal{M})$ where $\text{th} \triangleq \text{th}_f, \text{th}' \triangleq \text{th}_{f'}$, etc. All conditions in the column must be satisfied for the transition in the top row to be valid.

We prove the two threads satisfy the following specifications for any n:

$$\boxed{\text{EO\_inv}(\gamma_e, \gamma_o, \ell)}^{\mathcal{N}_{eo}} * \zeta \mapsto \{\text{even} := 42\} * \circ_{\gamma_e}(n) \twoheadrightarrow \text{wp}_\top^\zeta \text{ incr\_loop } \ell\ n\ \{\zeta \mapsto \emptyset\}$$

$$\boxed{\text{EO\_inv}(\gamma_e, \gamma_o, \ell)}^{\mathcal{N}_{eo}} * \zeta \mapsto \{\text{odd} := 42\} * \circ_{\gamma_o}(n) \twoheadrightarrow \text{wp}_\top^\zeta \text{ incr\_loop } \ell\ n\ \{\zeta \mapsto \emptyset\}$$

The proof follows similarly to the one of §4.2.1. We similarly open the invariant around the **cas** operation and in either case, whether it succeeds or fails, we take the appropriate model step.

4.3 Obtaining the Fairis Adequacy Theorem

This section describes in more detail how the Fairis logic and its adequacy theorem are obtained. In particular, we describe the critical Fuel-construction which, given a model $\mathcal{M}$, yields a Trillium model $\text{Fuel}(\mathcal{M})$ that handles fairness preservation and finite stuttering. The Fairis logic is then obtained by instantiating Trillium with this model and a well-chosen trace interpretation predicate S and relation ξ_{fuel}.

The Fuel-construction. Recall that a state f of $\text{Fuel}(\mathcal{M})$ has two components: a state $f.m$ of the fairness model $\mathcal{M}$, and a partial function $f.\text{map} : \text{ThreadId} \rightharpoonup (\text{Role}(\mathcal{M}) \rightharpoonup \mathbb{N})$ associating roles to thread ids and fuels (natural numbers) to roles. In actuality, not all such pairs are states of $\text{Fuel}(\mathcal{M})$, as states must satisfy certain conditions. Each role must be associated to at most one thread id:

$$\forall \zeta_1 \neq \zeta_2.\ \text{dom}(f.\text{map}(\zeta_1)) \cap \text{dom}(f.\text{map}(\zeta_2)) = \emptyset$$

and each *live role* must be associated to a thread id:

$$\forall \rho \in \text{lroles}(f.m).\ \exists \zeta \in \text{ThreadId}.\ \rho \in \text{dom}(f.\text{map}(\zeta))$$

where the live roles $\text{lroles}(m)$ of a state $m \in \mathcal{M}$ are defined as the set $\{\rho \mid \exists m',\ m \xrightarrow{\rho} m'\}$ of roles that can take a step from m. Intuitively, to prove fairness of a model trace which refines a fair program execution, we need to show that any live role eventually takes a step, so we need to relate *all* live roles to locales. We write $\text{roles}(f)$ for the set of roles allocated in $f.\text{map}$, formally, $\biguplus_{\zeta \in \text{dom}(f.\text{map})} \text{dom}(f.\text{map}(\zeta))$.

Roles that are in $\text{roles}(f) \setminus \text{lroles}(f.m)$ have a specific purpose: threads must be associated to a role to be allowed to take steps. It is often the case that a thread has logically terminated (in that it took all the steps corresponding to steps in the model) but has not actually terminated. This flexibility allows the thread to terminate in a bounded number of steps without cluttering the model.

We now turn to the more delicate aspect of the Fuel-construction: the transitions. There are two types of transitions: silent stuttering steps are labeled with tau_ζ, where ζ is meant to be the locale of the thread that takes the stuttering step, and visible steps, corresponding to the underlying model

steps which are labeled with $\text{vis}_\zeta(\rho)$ annotated with the local ζ taking the step, and the underlying role ρ in $\mathcal{M}$. The locales in labels are necessary to relate $\text{Fuel}(\mathcal{M})$ steps with the map f.th.

Figure 10 presents the list of conditions that define each type of transition. The conditions are stated using two auxiliary maps $\text{fm}_f : \text{Roles} \rightharpoonup \mathbb{N}$ associating role with its fuel, and $\text{th}_f : \text{Roles} \rightharpoonup \text{ThreadId}$ associating each role to some thread id. These two maps are derived from f.map as follows:

$$\text{fm}_f(\rho) = f \iff \exists \zeta, fs.\ \text{f.map}(\zeta) = fs \wedge fs(\rho) = f \qquad \text{th}_f(\rho) = \zeta \iff \rho \in \text{dom}(\text{f.map}(\zeta))$$

Condition ST restricts the evolution of the underlying fairness model, TH relates the thread with the state; DOM restricts the evolution of the domain: the only new roles are the roles which became live in the new underlying state (if any); DEC states that all threads associated to the thread that took the step must decrease their fuel, except possibly for the role that took a step, NI states that roles that changed theads must decrease their fuel; NI states that all other roles cannot increase their fuel, and finally REF restricts the new fuel of the new roles and of the role that took the step to the global bound F_{cap}. This last condition is necessary to restrict the branching of the model $\text{Fuel}(\mathcal{M})$, to be able to use the adequacy theorem of Trillium. It may not be obvious why condition CH is necessary: otherwise two threads could, at each step, take visible steps with roles ρ_1 and ρ_2 and, during that step, and change ownership of ρ_3, whose fuel could remain constant according to condition ND.

Logical Resources and the Trace Interpretation. The logical predicate $\zeta \Mapsto fs$ reflects a coherent view of the data in the $\text{Fuel}(\mathcal{M})$ model, as explained below.

To define the $\Mapsto$ predicate we use Iris's support for custom ghost state. These predicates satisfy, among other things, the following rules:

FUELMAP-AGREE
$$\frac{FuelMap(FM) \qquad \zeta \Mapsto fs}{FM(\zeta) = fs}$$

FUELMAP-UPDATE
$$\frac{FuelMap(FM) \qquad \zeta \Mapsto fs}{\Mapsto FuelMap(FM[\zeta := fs']) * \zeta \Mapsto fs'}$$

The link between the state of the Fuel-model and these resources, as well as between locales recorded in the Fuel-model and the program's locales, is specified in the trace interpretation $S(\tau, \kappa)$. First, τ and κ must have the same size, and their corresponding labels must match: the ith label ζ_i of τ is equal to $locale(l_i)$. Here l_i is the ith label of κ, and $locale$ extracts the locale from a label of $\text{Fuel}(\mathcal{M})$, *i.e.*, $locale(\text{tau}_\zeta) = locale(\text{vis}_\zeta(\rho)) = \zeta$. Second, the respective last states (tp, σ) and (m, fm) of τ and κ satisfy the following Iris predicate:

$$\exists FM.\ FuelMap(FM) * \bullet_{\gamma_\mathcal{M}} \text{m} * coherent(FM, \text{m}, \text{fm}) * \text{dom}(FM) \subseteq \text{dom}(\text{tp}) * ownHeap(\sigma)$$

where $ownHeap(\sigma)$ is the usual state interpretation of the Iris program logic reflecting the program's heap into Iris resources. Furthermore, $coherent(FM, \text{m}, \text{fm})$ is a predicate capturing that the ghost fuel map is coherent with respect to the model state. In particular, coherence captures that the ghost fuel map and model fuel map have the same locales, and that any role present in the former is also present in the latter with a larger or equal fuel. Additionally, coherence captures that any live role of the model exists in the ghost fuel map.

Adequacy. To use Trillium's adequacy theorem, the last missing piece is to choose the relation $\xi_{fuel} = \text{Fuel}(\xi)$. It turns out that, for our purpose, this relation can be rather weak. Given a finite program execution $\tau = (\text{tp}_1, \sigma_1) \xrightarrow{\zeta_1} (\text{tp}_2, \sigma_2) \xrightarrow{\zeta_2} \cdots \xrightarrow{\zeta_{n-1}} (\text{tp}_n, \sigma_n)$, and a finite $\text{Fuel}(\mathcal{M})$-trace $\kappa = (\text{m}_1, \text{map}_1) \xrightarrow{l_1} (\text{m}_2, \text{map}_2) \xrightarrow{l_2} \cdots \xrightarrow{l_{n-1}} (\text{m}_n, \text{map}_n)$, we define $\xi_{fuel}(\tau, \kappa)$ as follows:

$$\xi_{fuel}(\tau, \kappa) \triangleq \xi(\tau, \kappa) \wedge$$
$$\left(\forall j,\ \zeta_j = locale(l_j)\right) \wedge \tag{1}$$

$$(\forall i,\ \mathrm{rng}(\mathrm{th}_{\mathsf{f}_i}) \subseteq \mathrm{dom}(\mathrm{tp}_i) \wedge \forall \zeta, \rho,\ \mathrm{th}_{\mathsf{f}_i}(\rho) = \zeta \mathrm{tp}_i[\zeta] \text{ is a value} \Rightarrow \rho \notin \mathrm{lroles}(\mathsf{m}_i)) \quad (2)$$

We can now use Trillium's adequacy theorem (Theorem 3.2) to prove Fairis's adequacy theorem (Theorem 4.1) given the following lemma.

LEMMA 4.2. *The following holds for the parameters we have chosen:*

$$AlwaysHolds(\mathrm{Fuel}(\xi), c, \tilde{m}) \implies AlwaysHolds(\xi, c, m)$$

PROOF SKETCH. Let us consider a program execution τ and a model trace κ. It suffices to consider their last states, which we write respectively as (tp, σ) and f. That $\mathrm{rng}(\mathrm{th_f}) \subseteq \mathrm{dom}(\mathrm{tp})$ follows from *coherent*(FM, f.m, fm$_\mathsf{f}$) and from the fact that $\mathrm{dom}(\mathrm{th_f}) = \mathrm{dom}(\mathrm{fm_f})$. We prove the contrapositive of the second conjunct of (2) above. Assume a locale ζ and a role ρ such that $\mathrm{th_f}(\rho) = \zeta$ and such that $\mathrm{tp}[\zeta]$ is a value. Since we know all the postconditions of terminated threads hold, we know $\zeta \mapsto \emptyset$, which means, according to *coherent*(FM, f.m, fm$_\mathsf{f}$), that ρ cannot be live in f.m. The condition (1) on labels follows directly from the trace interpretation. □

Fairness Preservation. The adequacy theorem of Fairis gives us that the intial states of the program and of the Fuel($\mathcal{M}$) model are related: $(e, \sigma) \precsim_{\xi_{fuel}} \tilde{m}$. This refinement is useful because it has good properties: preservation of fairness and preservation of termination. To make things precise, we define some operations on traces: Any finite or infinite trace κ of Fuel($\mathcal{M}$) induces a trace $\overline{\kappa}$ in $\mathcal{M}$ which we obtain by removing tau transitions and projecting out the m component. Since there can only be finitely many tau transitions in a row (any such transition decreases the sum of all fuels), κ is finite if and only if $\overline{\kappa}$ is finite. And since $\hat{\xi}_{fuel}$ relates traces of the same length, if $\overline{\kappa}$ is finite and if $\hat{\xi}_{fuel}(\tau, \kappa)$ holds, then τ is finite.

Let us now explain why fairness is preserved in the other direction: if $\hat{\xi}_{fuel}(\tau, \kappa)$ and τ is fair, then $\overline{\kappa}$ is fair. We endow traces of Fuel($\mathcal{M}$) with the natural notion of fairness where we only look at the *roles* of the transitions, not the locales. It is obvious that if κ is fair then $\overline{\kappa}$ is fair, as tau-transitions do not play any role in fairness. Therefore, preservation of fairness boils down to:

LEMMA 4.3. *If τ is fair and if $\hat{\xi}_{fuel}(\tau, \kappa)$, then κ is fair.*

PROOF SKETCH. The proof is quite technical and consists of two nested inductions, but the idea is the following: consider some state f of κ and ρ which is live in that state, and call $\zeta = \mathrm{th_f}(\rho)$ its associated locale. We need to prove there eventually exists a ρ transition in κ. Since we have a fixed fair program trace τ, there exists n such that the next step of ζ is in at most n steps. We proceed by induction over $(\mathrm{fm_f}(\rho), n)$, ordered lexicographically.

Without loss of generality, we can assume it is the first state of κ. Write ζ for $\mathrm{th_f}(\rho)$. Consider the first step of the program with locale ζ'. There are two cases: (1) $\zeta \neq \zeta'$. Then we apply the induction hypothesis with the next state, the same fuel, and $n - 1$. (2) $\zeta = \zeta'$. If it is a ρ-transition we conclude. Otherwise, the fuel associated to ρ decreases, and we apply the induction hypothesis with the next state, a smaller fuel and some arbitrary n' obtained as above. □

5 THE ANERIS LOGIC

To reason about distributed systems, we instantiate Trillium with AnerisLang, the programming language accompanying Aneris, a higher-order distributed separation logic [Krogh-Jespersen et al. 2020]. AnerisLang is an OCaml-like programming language with network primitives for creating (`socket`) and binding (`socketbind`) network sockets as well as sending (`sendto`) and receiving (`receivefrom`) messages. The operational semantics of AnerisLang is designed so that the primitives closely model Unix sockets and UDP (unreliable) networking.

The Aneris instantiation of Trillium is conceptually simple as we will target safety trace properties. This means that we can "bake-in" the reflexive closure of the model and hence freely allow model stuttering. The result is a program logic and reasoning principles that are virtually identical to the original (non-relational) Aneris program logic, where the only difference is the addition of a single rule (ANERIS-TAKE-STEP) that allows us to relate an atomic step of the program to a corresponding step in the LTS model.

$$\frac{\text{ANERIS-TAKE-STEP}}{\text{wp}_{\mathcal{E}}^{\zeta}\, e\, \{v.\, \circ_{\gamma_{\mathcal{M}}}(\delta') \twoheadrightarrow \Phi(v)\} \qquad \circ_{\gamma_{\mathcal{M}}}(\delta) \qquad \delta \rightharpoonup_{\mathcal{M}} \delta' \qquad \text{Atomic}(e) \qquad e \notin \mathit{Val}}{\text{wp}_{\mathcal{E}}^{\zeta}\, e\, \{\Phi\}}$$

While fairly simple, this instantiation will still allow us to prove interesting properties. In what follows, we will show how we use Aneris to transport safety (trace) properties of TLA$^+$ protocol models to distributed programs that implement them. In the Appendix we also show how we, under reasonable liveness assumptions, can prove *eventual consistency* [Vogels 2009] of a Conflict-Free Replicated Data Type [Shapiro et al. 2011]. We leave a more principled approach to proving liveness properties of distributed systems as future work.

We use the Aneris instantiation of Trillium to show an intensional refinement between implementations of two classical distributed algorithms, Two-Phase Commit (TPC) [Gray 1978] and Single-Decree Paxos (SDP) [Lamport 1998, 2001], and their TLA$^+$ [Lamport 1992] models. As simple corollaries of the refinement, we show using a *single* modular specification (1) that clients are *safe*, *i.e.*, they do not crash, (2) a formal proof that the implementation *correctly* implements a protocol, and (3) correctness of the implementation by leveraging existing correctness properties of the models. The TLA$^+$ specification of TPC and the TLA$^+$ specification of SDP can both be found in the official TLA$^+$-examples repository on GitHub. In our formalization, we have manually translated the TLA$^+$ protocol specifications into STSs in Coq and proved their correctness properties.[8]

Note that correctness of the implementations *can* be established using regular Iris ghost resources and invariants, but doing so through a refinement has the immediate benefit that the necessary ghost theory is much simpler. The protocol logic is already encoded in the model and we "just" need to map the state of the model to the physical state of the distributed system. The only place we will need more sophisticated ghost theory is where the model is underspecified, *e.g.*, in how SDP distributes ballots among proposers. Additionally, by showing an intensional refinement, we show that the implementation of, say, SDP *actually* implements the SDP protocol and not just any other consensus protocol. While we do not show this explicitly, it also means that it is possible to transfer other trace properties of the model to the implementation.

Both the implementation, the model, and the refinement proof for the TPC protocol can be found in the Appendix. The development follows the same methodology as for SDP, which we describe below; we omit network- and state-related Aneris resources and focus on the core parts relevant for showing the refinement.

Single-Decree Paxos. The Paxos algorithm is a consensus protocol and its single-decree version allows a set of distributed nodes to reach agreement on a single value by communicating through message-passing over an unreliable network.

In SDP, each node in the system adopts one or more of the responsibilities of either *proposer*, *acceptor*, or *learner*. A value is chosen when a learner learns that a *quorum* (*e.g.*, a majority) of acceptors have accepted a value proposed by some proposer. The algorithm works in two phases: in the first phase, a proposer tries to convince a quorum of acceptors to promise that they will later accept its value. If it succeeds, it continues to the second phase where it asks the acceptors to

---

[8]A user who does not aim to be as foundational could, however, trust the translation and the existing TLA$^+$ proofs.

fulfill their promise and accept its value. To satisfy the requirements of consensus, each attempt to decide a value is distinguished with a unique totally-ordered round number or *ballot*. Each acceptor stores its current ballot and the last value it might have accepted, if any. Acceptors will only give a promise to proposers with a ballot greater than their current one, and in that case they switch to the proposer's ballot; proposers only propose values that ensure consistency, if chosen. By observing that a quorum of acceptors have accepted a value for the same ballot, learners will learn that a value has been chosen. We refer to Lamport [2001] for an elaborate textual description of the protocol.

Model. The TLA$^+$ model of SDP is summarized in Figure 11. The model is parameterized over a set of acceptors, Acceptor, and a type of values, Value, among which values are chosen. The state of the model consists of a set of sent messages $\mathcal{S} \in \mathscr{P}(\text{PaxosMessage})$ and two maps $\mathcal{B} : \text{Acceptor} \rightarrow \text{Option}(\text{Ballot})$ and $\mathcal{V} : \text{Acceptor} \rightarrow \text{Option}(\text{Ballot} \times \text{Value})$ that for each acceptor record the greatest ballot promise and the last accepted value together with its ballot, respectively. The message type is defined using a datatype-like notation as

$$\text{PaxosMessage} \triangleq \text{msg1a}(b) \mid \text{msg1b}(a, b, o) \mid \text{msg2a}(b, v) \mid \text{msg2b}(a, b, v)$$

where $a \in \text{Acceptor}$, $b \in \text{Ballot}$, $v \in \text{Value}$, and $o \in \text{Option}(\text{Ballot} \times \text{Value})$.

$$Q1bv(\mathcal{S}, Q, b) \triangleq \{m \in \mathcal{S} \mid \exists a, v.\, m = \text{msg1b}(a, b, \text{Some}(v)) \wedge a \in Q\}$$

$$HavePromised(\mathcal{S}, Q, b) \triangleq \forall a \in Q.\, \exists m \in \mathcal{S}, o.\, m = \text{msg1b}(a, b, o)$$

$$IsMaxVote(\mathcal{S}, Q, b, v) \triangleq \exists m_0 \in Q1bv(\mathcal{S}, Q, b), a_0, b_0.\, m = \text{msg1b}(a_0, b, \text{Some}(b_0, v)) \wedge$$
$$\forall m' \in Q1bv(\mathcal{S}, Q, b).$$
$$\exists a', b', v'.\, m' = \text{msg1b}(a', b, \text{Some}(b', v')) \wedge b_0 \geq b'$$

$$ShowsSafeAt(\mathcal{S}, Q, b, v) \triangleq HavePromised(\mathcal{S}, Q, b) \wedge (Q1bv(\mathcal{S}, Q, b) = \emptyset \vee IsMaxVote(\mathcal{S}, Q, b, v))$$

SDP-Phase1a

$$\frac{}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightharpoonup_{\text{SDP}} \mathcal{S} \cup \{\text{msg1a}(b)\}, \mathcal{B}, \mathcal{V}}$$

SDP-Phase1b

$$\frac{\text{msg1a}(b) \in \mathcal{S} \qquad b > \mathcal{B}(a) \qquad \mathcal{V}(a) = o}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightharpoonup_{\text{SDP}} \mathcal{S} \cup \{\text{msg1b}(a, b, o)\}, \mathcal{B}[a \mapsto \text{Some}(b)], \mathcal{V}}$$

SDP-Phase2a

$$\frac{\nexists v'.\, \text{msg2a}(b, v') \in \mathcal{S} \qquad Quorum(Q) \qquad ShowsSafeAt(\mathcal{S}, Q, b, v)}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightharpoonup_{\text{SDP}} \mathcal{S} \cup \{\text{msg2a}(b, v)\}, \mathcal{B}, \mathcal{V}}$$

SDP-Phase2b

$$\frac{\text{msg2a}(b, v) \in \mathcal{S} \qquad b \geq \mathcal{B}(a)}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightharpoonup_{\text{SDP}} \mathcal{S} \cup \{\text{msg2b}(a, b, v)\}, \mathcal{B}[a \mapsto \text{Some}(b)], \mathcal{V}[a \mapsto \text{Some}(b, v)]}$$

Fig. 11. TLA$^+$ specification of single-decree Paxos (SDP).

The SDP-Phase1a transition adds a $\text{msg1a}(b)$ message to the set of sent messages; this corresponds to the proposer asking the acceptors to *not* accept values for ballots smaller than b. If a $\text{msg1a}(b)$ message has been sent and b is greater than acceptor a's current ballot $\mathcal{B}(a)$ then the SDP-Phase1b transition updates a's state and sends a $\text{msg1b}(a, b, o)$ message where o is a's last accepted value, if any. This corresponds to an acceptor responding to a proposer's promise request.

The second phase is initiated using the SDP-Phase2a transition that corresponds to the proposer proposing a value v for ballot b by sending a $\text{msg2a}(b, v)$ message. However, the transition can only be made if no value has previously been proposed for ballot b and if a quorum Q of acceptors exists

Listing 1. Acceptor implementation.

```
let acceptor learners addr =
  let skt = socket () in
  socketbind skt addr;
  let maxBal = ref None in
  let maxVal = ref None in
  let rec loop () =
    let (m, sndr) = receivefrom skt in
    match acceptor_deser m with
    | inl bal =>
      if !maxBal = None ||
          Option.get !maxBal < bal then
        maxBal := Some bal;
        sendto skt
          (proposer_ser (bal, !maxVal)) sndr
      else ()
    | inr (bal, v) =>
      if !maxBal = None ||
          Option.get !maxBal <= bal then
        maxBal := Some bal;
        maxVal := Some accept;
        sendto_all skt learners
          (learner_ser (bal, v))
      else ()
  end; loop () in loop ()
```

Listing 2. Proposer implementation.

```
let proposer acceptors skt bal v =
  sendto_all skt acceptors
    (acceptor_ser (inl bal));
  let majority =
    (Set.cardinal acceptors) / 2 + 1 in
  let promises =
    recv_promises skt majority bal in
  let max_promise =
    find_max_promise promises in
  let av = Option.value max_promise v in
  sendto_all skt acceptors
    (acceptor_ser (inr (bal, av)))
```

Listing 3. Client implementation.

```
let client addr =
  let skt = socket () in
  socketbind skt addr;
  let (m1, sndr1) = receivefrom skt in
  let (_, v1) = client_deser m1 in
  let (m2, _) = wait_receivefrom skt
    (fun (_, sndr2), sndr2 <> sndr1) in
  let (_, v2) = client_deser m2 in
  assert (v1 = v2); v1.
```

such that the *ShowsSafeAt*$(\mathcal{S}, Q, b, v)$ predicate holds; this predicate is at the heart of the Paxos algorithm. Intuitively, the predicate holds if all acceptors in Q have promised not to accept values for any ballot less than b (*HavePromised*$(\mathcal{S}, Q, b)$) and *either* none of the acceptors have accepted any value for all ballots less than b *or* v is the value of the largest ballot that acceptors from Q have accepted. Following the SDP-Phase2b transition, acceptor a may accept a proposal for value v and ballot b by sending a msg2b(a, b, v) message and updating its state to reflect this fact. A value v has been chosen when a quorum of acceptors have sent a msg2b(a, b, v) message for some ballot b:

$$Chosen(\mathcal{S}, v) \triangleq \exists b, Q.\ Quorum(Q) \wedge \forall a \in Q.\ \text{msg2b}(a, b, v) \in \mathcal{S}$$

As follows from the theorem below, it is not possible for the protocol to choose two different values at the same time and hence SDP solves the consensus problem.

THEOREM 5.1 (CONSISTENCY, SDP MODEL). *Let* $\iota_{\text{SDP}} = (\emptyset, \lambda\_.\ None, \lambda\_.\ None)$. *If* $\iota_{\text{SDP}} \rightarrow^*_{\text{SDP}}$ $(\mathcal{S}, \mathcal{B}, \mathcal{V})$ *and both* $Chosen(\mathcal{S}, v_1)$ *and* $Chosen(\mathcal{S}, v_2)$ *hold then* $v_1 = v_2$.

Implementation. Listing 1 and Listing 2 show implementations of the acceptor and proposer roles, respectively. The learner implementation and utility functions such as recv_promises and find_max_promise are found in the Appendix.

The acceptor implementation receives as input a set of learner socket addresses and an address to communicate on. It creates a fresh socket, binds it to the address, and allocates two local references to keep track of its current ballot and last accepted value. In a loop, it listens for the two different kinds of messages that it may receive from the proposers. Given a phase one message, it only considers the message if the ballot is greater than its current ballot in which case it responds with its last accepted value. Given a phase two message, it only considers the message if the ballot is greater than or equal to its current ballot in which case it accepts the value and broadcasts the fact to all the learners. The learner implementation (included in the Appendix) simply waits for such a message for the same ballot from a majority of acceptors.

The proposer implementation receives as input a set of acceptor socket addresses, a bound socket, a ballot number and a value to (possibly) propose in the ballot. First phase is initiated by sending a message to all the acceptors and after receiving a response from a majority of the acceptors it continues to the second phase. In the second phase it picks the value of the maximum ballot among the responses; if no such value exist, it picks its own. The candidate is finally sent to all acceptors.

Note that this proposer implementation only proposes a value for a single ballot; typically, proposers will issue new ballots when learning that no decision has been reached due to messages being dropped or nodes crashing. Moreover, it is crucial that proposers do not issue proposals for the same ballot. In our Coq formalization, proposer p repeatedly issues new ballots of the form $k \cdot |\mathsf{Proposer}| + p$ for $k \in \mathbb{N}$ by keeping track of the last issued k in a local reference.

Consensus by Refinement. To show that the SDP implementation refines the SDP model we instantiate the Aneris logic with the model; the key part of the proof is to keep the $\circ_{\gamma_M}(\delta)$ resource in a global invariant that ties together the model state and the physical state with enough information to verify the implementation and for the refinement relation established through the adequacy theorem to be strong enough for proving our final correctness theorem (Corollary 5.2). Under this invariant we will *modularly* verify each Paxos role and each component in isolation.

To state the invariant, we use three kinds of resources corresponding to:

(1) sets of messages with predicates $\mathsf{Msgs}_\bullet(\mathcal{S})$ and $\mathsf{Msgs}_\circ(m)$ such that

$$\mathsf{Msgs}_\bullet(\mathcal{S}) * \mathsf{Msgs}_\circ(m) \vdash m \in \mathcal{S}$$
$$\mathsf{Msgs}_\bullet(\mathcal{S}) \vdash \Rrightarrow \left(\mathsf{Msgs}_\bullet(\mathcal{S} \cup m) * \mathsf{Msgs}_\circ(m)\right)$$

(2) maps, *e.g.*, with predicates $\mathsf{MaxBal}_\bullet(\mathcal{B})$ and $\mathsf{MaxBal}_\circ(a, b)$ such that

$$\mathsf{MaxBal}_\bullet(\mathcal{B}) * \mathsf{MaxBal}_\circ(a, b) \vdash \mathcal{B}(a) = b$$
$$\mathsf{MaxBal}_\bullet(\mathcal{B}) * \mathsf{MaxBal}_\circ(a, b) \vdash \Rrightarrow (\mathsf{MaxBal}_\bullet(\mathcal{B}[a \mapsto b']) * \mathsf{MaxBal}_\circ(a, b'))$$

(3) ballots with predicates $\mathsf{pending}(b)$ and $\mathsf{shot}(b, v)$ such that

$$\mathsf{pending}(b) * \mathsf{shot}(b, v) \vdash \mathsf{False} \qquad\qquad \mathsf{pending}(b) \vdash \Rrightarrow \mathsf{shot}(b, v)$$
$$\mathsf{pending}(b) * \mathsf{pending}(b) \vdash \mathsf{False} \qquad \mathsf{shot}(b, v_1) * \mathsf{shot}(b, v_2) \vdash v_1 = v_2$$

Equipped with these resource we can state the invariant:

$$I_{\mathsf{SDP}} \triangleq \exists \mathcal{S}, \mathcal{B}, \mathcal{V}. \circ_{\gamma_M}(\mathcal{S}, \mathcal{B}, \mathcal{V}) * \mathsf{Msgs}_\bullet(\mathcal{S}) * \mathsf{MaxBal}_\bullet(\mathcal{B}) *$$
$$\mathsf{MaxVal}_\bullet(\mathcal{V}) * BalCoh(\mathcal{S}) * MsgCoh(\mathcal{S})$$

The first part of the invariant ties the current state of the model $(\mathcal{S}, \mathcal{B}, \mathcal{V})$ to its logical *authoritative* counterparts which means that by owning a *fragmental* part you own a piece of the model: *e.g.*, by owning $\mathsf{MaxBal}_\circ(a, b)$ you may open the invariant and conclude $\mathcal{B}(a) = b$ where $\mathcal{B}$ is the current map of ballots. Intuitively, we will give acceptor a exclusive ownership of the parts of the model that should correspond to its local state (through resources $\mathsf{MaxBal}_\circ(a, b)$ and $\mathsf{MaxVal}_\circ(a, o)$). Similarly, by owning $\mathsf{Msgs}_\circ(m)$ one may conclude that the message m has in fact been added to the set of messages in the model; this predicate we will transfer when sending physical messages corresponding to m.

In the last part of the invariant, the $BalCoh(\mathcal{S})$ predicate simply requires that if $\mathsf{msg2a}(b, v) \in \mathcal{S}$ then $\mathsf{shot}(b, v)$ holds. This implies that by owning $\mathsf{pending}(b)$ you are the only entity that may propose a value for ballot b and it may never change. The $MsgCoh(\mathcal{S})$ predicate ties the physical state of the program to the model using Aneris-specific predicates for tracking the state of the network. This, for instance, forces acceptors and proposers to also add to the model state $\mathcal{S}$ any

message they send over the network. Hence, to verify a proposer or an acceptor that sends a message, the proof must open the invariant, use ANERIS-TAKE-STEP to take a step in the model, and update the corresponding logical resources to close the invariant. Following this methodology, we give specifications of the following shape to the proposer and acceptor components:

$$\boxed{I_{\mathsf{SDP}}} * \mathsf{MaxBal}_\circ\,(a, \mathsf{None}) * \mathsf{MaxVal}_\circ\,(a, \mathsf{None}) * \ldots \twoheadrightarrow \mathsf{wp}_\top^\zeta\ \mathtt{acceptor}\ L\ a\ \{\mathsf{False}\}$$

$$\boxed{I_{\mathsf{SDP}}} * \mathsf{pending}(b) * \ldots \twoheadrightarrow \mathsf{wp}_\top^\zeta\ \mathtt{proposer}\ A\ skt\ b\ v\ \{\mathsf{True}\}$$

omitting Aneris-specific network connectives in the precondition; the postcondition for `acceptor` may be False as it does not terminate. We give a similar specification to the learner. Working in a modular program logic, we can compose these specifications to get a single specification for a distributed system with both proposers, acceptors, and learners. By applying the adequacy theorem to this specification we get that the implementation indeed refines the TLA⁺ model of SDP.[9]

Consensus for the Implementation. Given the specification has been established for the implementation, we can state and prove that the consistency property holds for all executions by transporting the consistency property of the model. Let

$$ChosenI(M, v) \triangleq \exists b, Q.\ Quorum(Q) \wedge \forall a \in Q.\ \exists m \in M.\ m \sim \mathsf{msg2b}(a, b, v)$$

where M is a set of physical messages and $m \sim s$ holds when m is the serialization of the model message s. By picking a trace relation ξ_{SDP} that requires messages in the model to correspond to messages in the program state (as implied by $MsgCoh(\mathcal{S})$):

$$\xi_{\mathsf{SDP}}(\tau, \kappa) \triangleq \exists \mathcal{S}.\ last(\kappa) = (\mathcal{S}, \_, \_) \wedge messages(last(\tau)) \sim \mathcal{S} \wedge stuttering(\kappa)$$

we combine the adequacy theorem (Theorem 3.2) with our model correctness theorem (Theorem 5.1) to obtain the following corollary that *only* talks about the execution of the SDP implementation.

COROLLARY 5.2. *Let e be a distributed system obtained by composing n proposers, m acceptors, and k learners. For any T and σ, if $(e; \emptyset) \rightarrow^* (T; \sigma)$ and both $ChosenI(messages(\sigma), v_1)$ and $ChosenI(messages(\sigma), v_2)$ hold then $v_1 = v_2$.*

Functional Correctness. Corollary 5.2 is a meta-logic theorem (*e.g.*, in Coq) that only talks about the program execution and it follows from the adequacy theorem and the model correctness theorem. However, it is not only in the meta-logic that we can exploit properties of the model to prove properties about programs as the model is also embedded as a *resource* in the logic.

Listing 3 shows a client application that receives a message from two different Paxos learners and asserts that the two values are equal; if the two values do not agree, the program crashes. We can prove a specification for the client of the shape $\boxed{I_{\mathsf{SDP}}} * \ldots \twoheadrightarrow \mathsf{wp}\ \mathtt{client}\ a\ \{\ldots\}$. From the adequacy theorem it follows that the program is safe, *i.e.*, it does not crash, which means the asserted statement must always hold. In the proof of this specification, the client will receive ghost resources from the learners conveying that v_1 and v_2 have been chosen (*i.e.*, that a quorum of acceptors have accepted v_i). By opening the invariant I_{SDP} and hence obtaining the model resource $\circ_{\gamma_M}(\mathcal{S}, \mathcal{B}, \mathcal{V})$, we can combine this knowledge with Theorem 5.1—a property exclusively of the model—and conclude that $v_1 = v_2$. Naturally, we may still compose a distributed system containing the client together with proposers, acceptors, and learner nodes and derive a specification for the full system. This single specification for the full distributed system entails *both* the refinement of the TLA⁺ model and the safety of the programs running on all nodes.

---

[9]The full Coq proof amounts to about 1100 lines of proof scripts.

6 RELATED WORK

Refinement-based Verification of Distributed Systems. We focus on works that, as ours, aim at proving that concrete implementations refine models. The most closely related works are IronFleet [Hawblitzel et al. 2017] and Igloo [Sprenger et al. 2020]. In contrast to both, our approach is foundational: the operational semantics of the languages, the models, and the program logics are all formally defined in Coq, and through adequacy theorems of the program logic, the end result of a verification is a formal theorem expressed only in terms of the operational semantics of the programming language and the model.

IronFleet uses the Dafny verifier to verify implementations and encodes the refinement of an STS in preconditions and postconditions of programs but does not support node-local concurrency. IronFleet uses a pen-and-paper argument for proving liveness of simple programs (programs that consist of a simple event loop which calls terminating event handlers).

Igloo proves a particular kind of extensional safety properties. Igloo refines a high-level STS to a more low-level STS for each node of the system. STSs are annotated with IO operations which are used to generate IO specifications for network communications of the node in the style of Penninckx et al. [2015]. Programs are subsequently verified against this generated specification. The relationship between the implementation and the model considered in Igloo is a fixed relation, *i.e.* producing the same IO behavior. In contrast, our work allows an arbitrary intensional refinement relation to be specified and established between the program and the model.

Refinement in Iris. There has been earlier work on proving contextual refinements using Iris as discussed in the introduction. Additionally, Perennial [Chajed et al. 2019] defines correctness of a system using *concurrent recovery refinement*, requiring that the (possibly crashing) implementation and specification STS has the same external I/O. This notion of refinement is much coarser and does not allow you to prove, *e.g.*, fair termination. Tassarotti and Harper [2019] relates concurrent probabilistic programs to abstract specifications denoting indexed valuations, exhibiting a probabilistic coupling when assuming that the implementation terminates.

Our approach to termination-preserving refinement is similar in spirit to the one of Tassarotti et al. [2017] but applies to reasoning about refinement of general concurrent programs with respect to abstract models, not just compilation of session-typed programs. To the best of our knowledge, the expressiveness of the logics is roughly similar. The main difference is that Tassarotti et al. [2017] augments the Iris base logic with *linear* propositions, which requires modifying the definition of resource algebra to add a transition relation. We achieve similar results without heavy modifications, using that the authoritative state of the model is threaded through the weakest precondition, and by putting an exclusive structure on the set of roles owned by a thread, which prevents arbitrary weakening of the role resource, a limited form of linearity.

Besides efforts in Iris, Liang and Feng [2016, 2018] have also used refinement to show a wider range of liveness properties of concurrent programs, including programs with partial methods, but focusing on first-order logic and first-order programs. It would be interesting to investigate if Trillium could serve as a basis for generalizing the verification methods of Liang and Feng [2016, 2018] to higher-order logic and higher-order programs.

Simuliris [Gäher et al. 2022] is a separation logic for fair termination-preserving contextual refinements for concurrent program transformations that can exploit undefined behavior. In contrast to both Iris and Trillium, Simuliris is *not* step-indexed, and thus does not support impredicative invariants or higher-order ghost state, which we crucially target and rely on.[10]

---

[10]For example, higher-order ghost state is used to define Aneris's socket protocols.

Certified Abstraction Layers. A related approach to verification is *certified abstraction layers* [Gu et al. 2015], in particular their concurrent variant [Gu et al. 2018], which are used to verify the CertiKOS verified kernel [Gu et al. 2016]. Our approach is similar to that of CertiKOS in that both approaches use models to help verify programs. The main difference is that, in the CertiKOS approach, the person proving a refinement needs to work directly using the semantics of the program and of the model, which are both sets of traces. Our approach, on the other hand, is to use a program logic to do the heavy lifting of the refinement proof, which, we believe, lowers the proof burden dramatically. Another difference is that their notion of concurrent certified abstraction layers is more complex than our models, which are plain LTSs; but their models can be composed together.

Paxos Verification Efforts. Paxos and its multiple variants have been considered by many verification efforts using, *e.g.*, automated theorem provers and model checkers [Chand et al. 2016; Jaskelioff and Merz 2005; Kellomäki 2004; Maric et al. 2017; Padon et al. 2017]. These efforts all consider abstract *models* or specifications in high-level domain-specific languages of Paxos(-like) protocols and not actual implementations in a realistic and expressive programming language.

Kragl et al. [2020] work in the Boogie verifier and programming language where they express their high-level model, their low-level (non-distributed) imperative implementation, and the layers in between, all in Boogie. By contrast, our effort is foundational (in the technical sense of being formalized in Coq as mentioned above) and the implementation is carried out in the OCaml-like programming language AnerisLang with UDP network primitives.

García-Pérez et al. [2018] devise composable specifications for a pseudo-code implementation of Single-Decree Paxos and semantics-preserving optimizations to the protocol on pen-and-paper but without a formal connection to their implementation in Scala; it would be interesting future work to implement and verify the same optimizations in our setting.

7 CONCLUSION AND FUTURE WORK

In this paper, we explored how intensional refinement *is* indeed a viable methodology for strengthening higher-order concurrent and distributed separation logics to non-trivial safety and liveness properties using Trillium and its instantiations. We have developed Fairis, a higher-order concurrent separation logic, and we have shown how the logic gives us a methodology for proving liveness of concurrent programs under fair scheduling assumptions. Moreover, we instantiated Trillium with a distributed language and obtained an extension of Aneris, a distributed separation logic, that we have used to show refinement relations between distributed systems and their TLA$^+$ models.

Future work includes extending Trillium's support for modular reasoning to also allow specifications of library functions to be modular with respect to the model, such that a library function can be specified with respect to one model and client code can be specified with respect to another model in isolation. Currently, library functions can be reasoned about modularly in Trillium using higher-order specifications. For example, our Paxos implementation makes a call to the sendto_all function, whose specification quantifies over arbitrary socket protocols (themselves higher-order predicates), and in turn the sendto_all function internally makes a call to a function set_iter, whose specification uses impredicative quantification to support arbitrary callback functions. An important point here, however, is that these library specifications do not interact with the model, *i.e.*, execution corresponds to stuttering steps on the model side. A concrete goal would be to give a modular specification of a fair lock and then verify termination of a client that uses the lock, but only by relying on its specification.

Data availability statement. The Coq formalization accompanying this work is available on Zenodo [Timany et al. 2023] and on GitHub at https://github.com/logsem/trillium.

ACKNOWLEDGMENTS

REFERENCES

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for Free from Separation Logic Specifications. *Proc. ACM Program. Lang.* 5, ICFP, Article 81 (aug 2021), 29 pages. https://doi.org/10.1145/3473586

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.* 243–258. https://doi.org/10.1145/3341301.3359632

Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. 2016. Formal Verification of Multi-Paxos for Distributed Consensus. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9995)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). 119–136. https://doi.org/10.1007/978-3-319-48989-6_8

Rance Cleaveland and Oleg Sokolsky. 2001. CHAPTER 6 - Equivalence and Preorder Checking for Finite-State Systems. In *Handbook of Process Algebra*, J.A. Bergstra, A. Ponse, and S.A. Smolka (Eds.). Elsevier Science, Amsterdam, 391–424. https://doi.org/10.1016/B978-044482830-9/50024-2

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018.* 442–451. https://doi.org/10.1145/3209108.3209174

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2020. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Log. Methods Comput. Sci.* 17 (2020).

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. 2018. Paxos Consensus, Deconstructed and Abstracted. 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, 912–939. https://doi.org/10.1007/978-3-319-89884-1_32

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, Article 74 (apr 2022), 30 pages. https://doi.org/10.1145/3527318

Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle (Eds.). Lecture Notes in Computer Science, Vol. 60. Springer, 393–481. https://doi.org/10.1007/3-540-08755-9_9

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2023. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *CoRR* abs/2301.10061 (2023). https://doi.org/10.48550/arXiv.2301.10061 arXiv:2301.10061

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. ACM.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. ACM, 646–661.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM* 60, 7 (June 2017), 83–92. https://doi.org/10.1145/3068608

Koen Jacobs, Dominique Devriese, and Amin Timany. 2022. Purity of an ST Monad: Full Abstraction by Semantically Typed Back-Translation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 82 (apr 2022), 27 pages. https://doi.org/10.1145/3527326

Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully Abstract from Static to Gradual. *Proc. ACM Program. Lang.* 5, POPL, Article 7 (jan 2021), 30 pages. https://doi.org/10.1145/3434288

Mauro Jaskelioff and Stephan Merz. 2005. Proving the Correctness of Disk Paxos. *Arch. Formal Proofs* 2005 (2005). https://www.isa-afp.org/entries/DiskPaxos.shtml

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

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

Pertti Kellomäki. 2004. *An Annotated Specification of the Consensus Protocol of Paxos Using Superposition in PVS.* Technical Report. Tampere University of Technology. Institute of Software Systems.

Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive sequentialization of asynchronous programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020.* 227–242. https://doi.org/10.1145/3385412.3385980

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 205–217. https://doi.org/10.1145/3009837.3009855

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17).* Association for Computing Machinery, New York, NY, USA, 218–231. https://doi.org/10.1145/3009837.3009877

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.* 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

Leslie Lamport. 1992. Hybrid Systems in TLA$^+$. In *Hybrid Systems*, Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel (Eds.). Lecture Notes in Computer Science, Vol. 736. Springer, 77–102. https://doi.org/10.1007/3-540-57318-6_25

Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58.

Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. 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, 385–399. https://doi.org/10.1145/2837614.2837635

Hongjin Liang and Xinyu Feng. 2018. Progress of concurrent objects with partial methods. *Proc. ACM Program. Lang.* 2, POPL (2018), 20:1–20:31. https://doi.org/10.1145/3158108

Ognjen Maric, Christoph Sprenger, and David A. Basin. 2017. Cutoff Bounds for Consensus Algorithms. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 217–237. https://doi.org/10.1007/978-3-319-63390-9_12

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 108:1–108:31. https://doi.org/10.1145/3140568

Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 158–182. https://doi.org/10.1007/978-3-662-46669-8_7

Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Convergent and Commutative Replicated Data Types. *Bull. EATCS* 104 (2011), 67–88. http://eatcs.org/beatcs/index.php/beatcs/article/view/120

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211.* 80–95. https://doi.org/10.1145/3453483.3454031

Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David Basin. 2020. Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 152 (Nov. 2020), 31 pages. https://doi.org/10.1145/3428220

Joseph Tassarotti and Robert Harper. 2019. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 64:1–64:30. https://doi.org/10.1145/3290377

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Amin Timany and Lars Birkedal. 2019. Mechanized Relational Verification of Concurrent Programs with Continuations. *Proc. ACM Program. Lang.* 3, ICFP, Article 105 (July 2019), 28 pages. https://doi.org/10.1145/3341709

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2023. *Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement - Coq Artifact.* https://doi.org/10.5281/zenodo.10100892

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *PACMPL* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44. https://doi.org/10.1145/1435417.1435432