arXiv:2109.07863v1 [cs.PL] 16 Sep 2021

Trillium: Unifying Refinement and Higher-Order Distributed Separation Logic

AMIN TIMANY, Aarhus University, Denmark
SIMON ODDERSHEDE GREGERSEN, Aarhus University, Denmark
LÉO STEFANESCO, Collège de France, France
LÉON GONDELMAN, Aarhus University, Denmark
ABEL NIETO, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

We present a unification of refinement and Hoare-style reasoning in a foundational mechanized higher-order distributed separation logic. This unification enables us to prove formally in the Coq proof assistant that concrete implementations of challenging distributed systems refine more abstract models and to combine refinement-style reasoning with Hoare-style program verification. We use our logic to prove correctness of concrete implementations of two-phase commit and single-decree Paxos by showing that they refine their abstract TLA^{+} specifications. We further use our notion of refinement to transfer fairness assumptions on program executions to model traces and then transfer liveness properties of fair model traces back to program executions, which enables us to prove liveness properties such as strong eventual consistency of a concrete implementation of a Conflict-Free Replicated Data Type and *fair* termination of a concurrent program.

CCS Concepts: • **Theory of computation → Program verification**; **Distributed algorithms**; **Separation logic**.

1 INTRODUCTION

Refinement is an old idea and an established approach to reason about correctness of programs [Burstall and Darlington 1975; Dijkstra 1970; Gerhart 1975; Hoare 1969, 1972; Wirth 1971] and many variations of refinement have been studied using different formalisms. In the context of concurrent and distributed systems, one popular formalism is TLA^{+} [Lamport 1992], which is a formal specification language used to design, model, and verify concurrent and distributed systems. In TLA^{+}, systems are modelled using state transition systems and both safety and liveness properties are amenable to finite model checking. The tool suite has seen successful industrial use by companies like Intel [Beers 2008], Amazon [Newcombe 2014; Newcombe et al. 2015], and Microsoft [Lardinois 2017]. While the TLA^{+} verification system allows to uncover design flaws and to reason about the correctness of abstract system specifications, it offers no guarantees about an *implementation* of such a system nor about its relation to the abstract specification.

In this paper we show how to unify refinement and Hoare-style reasoning in a foundational mechanized higher-order distributed separation logic. This unification enables us to prove formally in the Coq proof assistant that concrete implementations of challenging distributed systems refine more abstract models of such, and to combine refinement-style reasoning with Hoare-style reasoning. We focus on *history-sentitive* refinement relations that relate *traces* of program executions to *traces* of a model. We use this history-sensitive notion of refinement to transfer fairness assumptions on program executions to model traces, and we demonstrate (in §4) how to make use of this to prove liveness properties, namely strong eventual consistency of a concrete implementation of a *Conflict-Free Replicated Data Type* (CRDT) [Shapiro et al. 2011], and fair termination

Authors' addresses: Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Simon Oddershede Gregersen, Aarhus University, Denmark, gregersen@cs.au.dk; Léo Stefanesco, Collège de France, France, leo.stefanesco@college-de-france.fr; 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.

of a concurrent program. A special case of history-sensitive refinement relations are standard refinement relations that relate program states to states of a model, and we use our logic to prove that concrete implementations of *two-phase commit* and *single-decree Paxos* formally refine their abstract TLA specifications (§3). As one would hope, we can use the refinements to transfer results about the TLA specifications (*e.g.*, consistency in Theorem 3.1) to results about the program implementations.

We develop our unification of refinement and Hoare-style reasoning by extending Aneris [Krogh-Jespersen et al. 2020], a higher-order distributed separation logic, with refinement-style reasoning. The Aneris logic, which is aimed at modular reasoning about distributed systems implementations, is itself an extension of the Iris [Jung et al. 2016, 2018, 2015; Krebbers et al. 2017a] higher-order concurrent separation logic. There has been earlier work on refinement in Iris. However, most of that work has focused on contextual refinement, where one (higher-order concurrent imperative, but not distributed) program is related to another program [Frumin et al. 2018; Krebbers et al. 2017b; Krogh-Jespersen et al. 2017; Spies et al. 2021; Timany et al. 2018] or termination-preserving refinements among programs [Spies et al. 2021; Tassarotti et al. 2017] (one exception is the work of Tassarotti and Harper [2019] who relate programs to abstract specification programs denoting indexed valuations). In contrast, we focus on establishing that a program refines a more abstract *model*, which then serves as a specification, allowing us, for instance, to prove that our Paxos implementation refines the TLA$^+$ specification of Paxos.

We follow the "Iris approach" and define a generic program logic, called Trillium, for establishing history-sensitive refinement of programs written in an arbitrary programming language specified by an operational semantics. Trillium is defined on top of the Iris base logic and plays a similar role as the generic Iris program logic for reasoning about functional correctness: Trillium has a notion of weakest precondition predicate (and associated Hoare triples) for which a number of generally applicable proof rules have been proved. Perhaps surprisingly, given that we wish to reason about refinements among traces, the weakest precondition predicate of Trillium is a predicate over program expressions (in contrast to whole execution traces). Thus Trillium supports *local reasoning* about history-sensitive refinement. Importantly it also means that the proof rules of Trillium include all the usual rules of the Iris program logic; there is just one additional proof rule for reasoning about refinement. Thus Trillium is a conservative extension of the Iris program logic and hence we can seamlessly reuse Iris program logic proofs in the context of Trillium. It also means that the *same* weakest precondition specification can be used to capture refinement as well as functional correctness. Thus weakest preconditions for refinements can, e.g., be used by clients to show their functional correctness, which might rely on a property obtained via the abstract model through the refinement. Users of Trillium can then instantiate it with their own choice of programming language and prove soundness of additional programming language-specific proof rules. In this paper, we mostly focus on an instantiation of Trillium with AnerisLang, the distributed higher-order concurrent imperative programming language of the Aneris program logic; this results in a unified logic for modular refinement and Hoare-style reasoning, which extends the earlier Aneris program logic. However, in §5 we consider another instantiation of Trillium with HeapLang, a concurrent (non-distributed) language, and show how Trillium can also be used to reason about termination of concurrent programs, by establishing *fair* termination-preserving refinement of a suitable model. This demonstrates (through an admittedly simple, but very tricky, example) how we may integrate some of the key ideas of Tassarotti et al. [2017]'s work on fair termination-preserving refinement of concurrent session-typed programs in an arguably simpler general program logic (not focused on compilation of session-typed programs) and without having to introduce linear predicates to the logic.

We emphasize that since our instantiation of Trillium with AnerisLang is an extension of Aneris, it inherits Aneris's support for *modular reasoning* and extends it to refinement reasoning. In particular, it supports both (1) *horizontal modularity* via node-local and thread-local reasoning, which allows one to verify distributed systems by verifying each thread and each node in isolation, and (2) *veritical modularity* via separation logic features such as the frame rule and the bind rule, which allow one to compose proofs of different components within each node. We leverage this support for modularity in our case studies, e.g., in the refinement proof of Paxos, where we specify and prove each component of the implementation in isolation. This is achieved by sharing the more global abstract model among the different components via an invariant. Moreover, we use a *single* specification for both proving that the implementation refines the Paxos model and to verify functional correctness of a client of the Paxos implementation; the proof of the client in turn relies on the consistency property of the Paxos model.

Contributions. In summary, we make the following contributions:

- We introduce Trillium, a generic program logic that unifies Hoare-style reasoning with local reasoning about history-sensitive refinement relations among traces of program executions and traces of a model (§2).
- We instantitate Trillium with AnerisLang, a distributed higher-order concurrent imperative programming language (similar to OCaml with sockets) to get an extension of Aneris that unifies refinement and Hoare-style reasoning about implementations of distributed systems.
- We use this instantiation to establish correctness of concrete implementations of two distributed protocols, two-phase commit and single-decree Paxos, by showing that they refine their abstract TLA$^+$ specifications. To the best of our knowledge, this is the first foundationally verified proof that a concrete implementation of a distributed protocol correctly implements its abstract TLA$^+$ specification. We also demonstrate how to use the *same* refinement specification both to prove properties about the implementation, by relying on existing correctness theorems of the TLA$^+$ specification, and to show functional correctness of client programs, thus leveraging the unification of refinement and Hoare-style reasoning (§3).
- We further show functional correctness and strong eventual consistency of a concrete implementation of a CRDT. The proof of the latter relies on the fact that we establish that the implementation is a history-preserving refinement of a model; this allows us to transfer fairness assumptions on the implementation to the model and prove strong 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; the concurrent interactions with the user-exposed operations makes it non-trivial to reason about eventual consistency.
- We instantiate Trillium with HeapLang, a higher-order concurrent imperative programming language and use the resulting logic to show fair termination of a concurrent program by establishing a fair termination-preserving refinement of a suitable model.

All the results presented in this paper have been formalized in Coq, on top of the Iris base-logic formalization and using the Iris Proof Mode [Krebbers et al. 2017b]. We use the Iris Proof Mode to support interactive verification in Trillium (in the same style as one has for Iris), which means that one can reason formally in Trillium much in the same way as one reasons in Coq. We stress that our formal development is what is often called foundational: the operational semantics of the distributed programming language, the abstract models, and the model of the program logic 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.

2 TRILLIUM: A TRACE PROGRAM LOGIC

In this section we present the core of Trillium which relates execution traces of programs written in an arbitrary programming language to traces over a model. For generality, Trillium does not fix the notion of model; we just consider model traces to be traces over some arbitrary set of auxiliary states. Before detailing Trillium we briefly discuss the general notion of programming language we will work with, and our conventions and notations for programs and auxiliary traces. Afterwards we present the logic and then briefly discuss how it is instantiated to reason about AnerisLang programs. We conclude this section with a discussion of adequacy (soundness) of Trillium.

Programming Languages Considered. Trillium is defined with respect to an abstract operational semantics for a (concurrent/distributed) programming language. We assume that the programming language comes with a set of expressions *Expr*, a set of values $Val \subseteq Expr$, a set of program states, *State* (an abstraction of the heap, the network state, *etc.*), and a step relation $\rightarrow$. The latter relates a pair of an expression and a state to a triple consisting of an expression, a state, and a (possibly empty) list of expressions, corresponding to the threads forked by the step of computation. These ingredients should satisfy standard requirements, *e.g.*, that values cannot take a step ($\forall v \in Val, \sigma \in State.\ (v, \sigma) \not\rightarrow$); see Jung et al. [2018] for details. The step relation is then lifted to *machine configurations*, *i.e.*, a pair of a thread pool (possibly across multiple machines in the case of distributed systems) and a state:

$$\frac{(e_i, \sigma) \rightarrow (e_i', \sigma', (e_{f_1}, \cdots, e_{f_k}))}{(e_1, \cdots, e_{i-1}, e_i, e_{i+1}, \cdots e_n; \sigma) \twoheadrightarrow (e_1, \cdots, e_{i-1}, e_i', e_{i+1}, \cdots e_n, e_{f_1}, \cdots, e_{f_k}; \sigma')}$$

The idea is that the machine takes a step whenever a thread takes a step, in which case the threads possibly forked off are added to the list of all threads of the machine. Another way the machine can take a step in our notion of programming language is through the system-step relation $\rightarrow_{\mathsf{sys}} \subseteq State \times State$, which a language should also specify along with its step relation $\rightarrow$. System steps are those that do not correspond to actual program execution steps; *e.g.*, in AnerisLang, a system step can correspond to the network dropping a message. Internal steps are also lifted to the machine level:

$$\frac{\sigma \rightarrow_{\mathsf{sys}} \sigma'}{(e_1, \cdots, e_n; \sigma) \twoheadrightarrow (e_1, \cdots, e_n; \sigma')}$$

We write c for a configuration and write $state(c)$ and $thread(i, c)$ for the state of c and the ith thread of c, respectively. Furthermore, we write $update_{th}(i, e, \sigma, c)$ for the configuration that is the same as c, except that the state is updated to σ and the ith thread is updated to be e. Similarly, we write $update_{st}(\sigma, c)$ for the configurationn that is the same as c except the state is updated to be σ.

We further require a programming language to be equipped with a set of *evaluation contexts.* As we shall see, we use those to express the relationship between a program at hand and the program execution trace. Evaluation contexts are required to satisfy the usual properties, *e.g.*, for a context K, if $(e, \sigma) \rightarrow (e', \sigma')$ then $(K[e], \sigma) \rightarrow (K[e'], \sigma')$; see Jung et al. [2018] for details.

Finite and Possibly-Infinite Traces. A *finite trace* (over some set) is a *non-empty* sequence (of elements from the set). A *possibly-infinite trace* (over some set) is a finite or infinite sequence (of elements from the set). Note that the empty sequence is a possibly-infinite trace. Whenever we mention the word trace in the rest of the paper, we mean a finite trace, unless explicitly stated otherwise. We let τ range over traces over the set of program configurations and let μ range over

traces over the set of auxiliary states. We use $\ddot{\tau}$ and $\ddot{\mu}$ to range over possibly-infinite traces of program configurations and auxiliary states respectively. Furthermore, we write $\mathit{first}(t)$ and $\mathit{last}(t)$ for the first and last element of a trace t, respectively—these are well-defined total functions as traces are required to be non-empty. We write $[a]_{tr}$ for the singleton finite trace consisting of only a, and $t ::_{tr} a$ for the extension of trace t with element a appended to it. We use list notations $[]$ and $a :: t$ (where t is a possibly-infinite trace) for possibly infinite traces. Do note that for finite traces, $::_{tr}$ adds an element at the end, whereas for possibly-infinite traces, $::$ adds an element at the front.[1] We write $\mathit{get}(i, t)$ for the ith element of the trace t and $\mathit{length}(t)$ for the length of the trace t.

An execution trace τ is valid, denoted $\mathit{ValidExec}(\tau)$, if every configuration in the trace makes a step in the operational semantics to the next configuration. Formally, we use the this inductive definition: $\mathit{ValidExec}(\tau) \triangleq (\exists c.\ \tau = [c]_{tr}) \vee (\exists c, \tau'.\ \tau = c ::_{tr} \tau' \wedge \mathit{ValidExec}(\tau') \wedge \mathit{last}(\tau') \twoheadrightarrow c)$.

2.1 The Program Logic Trillium

The core idea of Trillium is to track an auxiliary trace alongside the program execution trace and *enforce* that whenever the program takes a step, there is an extension of the auxiliary trace that corresponds to this step.

As indicated in the Introduction, Trillium is centered around the concept of a weakest precondition; Hoare triples can then, as usual, be defined in terms of weakest preconditions. The weakest precondition of Trillium is defined using the Iris base logic, similarly to how the weakest precondition of the Iris program logic is defined using the Iris base logic [Jung et al. 2018]. We recall that the Iris base logic is a higher-order logic equipped with facilities for definitions and reasoning by guarded recursion and with support for reasoning about ownership (separation logic) and invariants.

Concretely, the Trillium definition of the weakest precondition predicate $\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}$ is as follows:

$$\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\} \triangleq \begin{cases} \Rrightarrow_{\mathcal{E}} \Phi(e) & \text{if } e \in \mathit{Val} \\[1em] \begin{array}{l} \forall \tau, \mu, i, K. \\ \quad \mathit{thread}(i, \mathit{last}(\tau)) = K[e] \mathrel{-\!\!*} \\ \quad \mathit{ValidExec}(\tau) \mathrel{-\!\!*} \\ \quad \mathit{StateInterp}(\tau, \mu)\ {}^{\mathcal{E}}\!\!\Rrightarrow\!\!*^{\emptyset} \\ \quad \mathit{reducible}(e, \mathit{state}(\mathit{last}(\tau))) * \\ \quad \forall e', \sigma', (e_{f_1}, \cdots, e_{f_k}). \\ \qquad (e, \mathit{state}(\mathit{last}(\tau))) \rightarrow (e', \sigma', (e_{f_1}, \cdots, e_{f_k})) \mathrel{-\!\!*} \triangleright\ {}^{\emptyset}\!\!\Rrightarrow\!\!*^{\mathcal{E}} \\ \qquad \exists \delta. \mathit{StateInterp}(\tau ::_{tr} \mathit{update}_{th}(i, K[e'], \sigma', \mathit{last}(\tau)), \mu ::_{tr} \delta) * \\ \qquad\quad \mathsf{ValidEvolution}(\tau, \mu, \mathit{update}_{th}(i, K[e'], \sigma', \mathit{last}(\tau)), \delta) * \\ \qquad\quad \mathsf{wp}_{\mathcal{E}}\ e'\ \{\Phi\} * \mathop{\Large *}_{1 \le j \le k} \mathsf{wp}_{\top}\ e_{f_j}\ \{\mathsf{True}\} \end{array} & \text{otherwise} \end{cases}$$

Here the postcondition Φ is a predicate that takes a return value as an argument. We sometimes write Φ as $x.\ P$; then x acts as a binder for the return value in P. In the above definition, Iris-specific logical connectives are typeset in blue; to understand the high-level ideas of the definition and, in particular, what is new compared to the Iris definition of weakest preconditions, they can mostly

[1] As will become clearer later, this difference in notation is useful because we use finite traces to capture what has happened in the past (they "grow at the end" when time passes), whereas we use possibly-infinite traces to capture the future (they shrink when time passes).

be ignored. On a first reading $\Rrightarrow_{\mathcal{E}}$ and $\triangleright$ may be ignored, and $*$ may be thought of as ordinary conjunction, and both $-\!\!*$ and ${}^{\mathcal{E}}\!\Rrightarrow\!\!*^{\mathcal{E}'}$ as ordinary implication.

Considering that Trillium is a program logic designed for relational reasoning, perhaps the most surprising fact about the definition is its overall form, *i.e.*, that $\mathsf{wp}\ e\ \{\Phi\}$ only involves a program expression e and a predicate Φ on program values; and in particular, that it does not explicitly refer to auxiliary state. This is because the relationship between the program and the auxiliary state is encapsulated inside the definition of the weakest precondition. This makes the definition very flexible: when a user instantiates Trillium they can decide how to set up the relationship between traces of the program and traces of the auxiliary state, and how to reflect this relationship in Iris resources—see §2.2 for a concrete example. Moreover, it also means that the Trillium weakest precondition is conservative over standard Iris and that all the Iris proof rules for Iris weakest preconditions still hold for Trillium.[2] Hence all proofs carried out in the standard Iris program logic can be straightforwardly adapted to Trillium.

Next we remark that, just as for standard Iris weakest preconditions, our definition implies safety (see §2.3 for more details): if $\mathsf{wp}\ e\ \{\Phi\}$ holds then e will not get stuck and whenever it reduces to a value, then that value satisfies the postcondition. This follows from the high-level pattern of the definition: either e is a value in which case the postcondition must hold, or e is reducible in the current state (the state of the last configuration in our execution trace τ) and, furthermore, whatever e reduces to in this current state (as well as all the forked threads) should again satisfy the weakest precondition—we do not care about the postcondition of the forked threads.

The parts typeset in red and green in the definition above are, respectively, the parts that have been added or adapted, compared to standard Iris weakest preconditions. We now explain these parts in more detail.

The state interpretation predicate, $\mathit{StateInterp}$, in Iris weakest preconditions takes only the current state as an argument; here it has been extended to take the entire execution trace of the program as well as the auxiliary trace as arguments. This is crucial and means in particular that we can reflect the history of the program execution in the program logic and use it in our reasoning. For example, in the AnerisLang instantiation of Trillium, it allows us to track the history of sent and received messages. In §4 we discuss how this is useful for expressing the correctness of CRDTs. The role of the $\mathit{StateInterp}$ predicate is to tie the state of both the program and the auxiliary state to Iris resources, *e.g.*, the points-to predicate of separation logic for heap locations; see Trillium's instantiation with AnerisLang in §2.2 for details. To appreciate the role of the $\mathit{StateInterp}$ predicate better, let us consider part of the soundness proof of the following Aneris inference rule:[3]

WP-LOAD

$$\frac{\ell \mapsto_{ip} v}{\mathsf{wp}_{\mathcal{E}}\ \langle ip;\ !\,\ell\rangle\ \{x.\ x = v * \ell \mapsto_{ip} v\}}$$

This rule states that if we own the location ℓ with value v on a node with IP address ip, *i.e.* we have the points-to predicate $\ell \mapsto_{ip} v$, then on that node, reading ℓ returns a value that is equal to v and, moreover, we retain ownership of ℓ. To show soundness of this rule, we need to show that the program $!\,\ell$ does not get stuck when run in the *current state* which would happen if ℓ was not allocated. Hence, we need to rule out that case and show that no matter what the current state may be, ℓ is allocated in that state—note how the current state (in our case part of τ) is universally quantified in the definition of the weakest precondition predicate. This is where $\mathit{StateInterp}$ plays an important role: $\mathit{StateInterp}$ and the points-to predicate are defined so that they satisfy the

[2] The only exception is that Trillium does not support Iris prophecies [Jung et al. 2020].

[3] In the AnerisLang instantion of Trillium, a program expression consists of a pair of a node IP address and an ordinary expression.

following property:[4]

$$StateInterp(\tau, \mu) * \ell \mapsto_{ip} v \vdash Heap(ip, last(\tau))(\ell) = v$$

where $Heap(ip, c)$ is the heap (a partial map with finite support from locations to values) for the node with IP address ip in the state of the configuration c. Here $\vdash$ is Iris's entailment relation. Hence, we can conclude that ℓ is allocated in the heap of the current state and that the program does not get stuck.

Apart from knowing that the resources pertaining to state, such as heap points-to predicates, network messages, *etc.*, are properly reflected in Iris resources, we need to know that the execution trace τ is consistent with a given program. That is, when proving wp $e \{\Phi\}$ we need to know that whatever the execution trace τ up to now is, the expression e is now about to take a step. This is captured by $thread(i, last(\tau)) = K[e]$, which states that on thread i of the current configuration, $last(\tau)$, the expression e can now take a step, *i.e.*, e appears under an evaluation context K.

The definition of the weakest precondition requires that after e takes a step, there must be an auxiliary state δ such that the following holds:

(1) The state interpretation holds for the resulting execution trace and auxiliary trace (note that both traces are extended with their respective new states).
(2) The user specified predicate ValidEvolution is satisfied.
(3) The weakest precondition holds for e', the program that e steps to.
(4) All threads forked during the step also satisfy the weakest precondition.

Note also that the user of Trillium can restrict the new auxiliary state δ by picking ValidEvolution appropriately.

In summary, the weakest precondition wp $e \{\Phi\}$ states that e is safe to execute (does not get stuck), that any value resulting from e satisfies the postcondition Φ, and, that every execution step has been matched by an auxiliary step.

That Trillium is conservative over the Iris program logic follows from the fact that we have proven all the lemmas used in Iris to facilitate the instantiation of it to different programming languages, *e.g.*, lemmas that allow deriving proof rules such as WP-LOAD above. As mentioned in the Introduction, Trillium has just one additional rule for reasoning about refinement, called WP-ATOMIC-TAKE-STEP. Intuitively, this rule states that when proving a weakest precondition for an atomic program (a program that takes at most one step) one can choose the auxiliary state after the step, provided that the ValidEvolution relation is satisfied. The general formulation of this rule involves many details (and is only intended to be used once when instantiating Trillium) and hence we omit it. Below we show and discuss the rule ANERIS-WP-ATOMIC-TAKE-STEP, which is a consequence of WP-ATOMIC-TAKE-STEP in the instantiation of Trillium with AnerisLang. The general rule, WP-ATOMIC-TAKE-STEP, can be found in the accompanying Coq development.

2.2 Aneris Instantiation of Trillium

AnerisLang is an ML-like programming language with network primitives for creating and binding sockets as well as sending (`sendto`) and receiving (`receivefrom`) messages on those sockets. In AnerisLang only strings can be sent over the network and hence programs need to appropriately (de)serialize values as necessary. The operational semantics is designed so that these networking primitives closely model Unix sockets and UDP networking. The operational semantics of AnerisLang is, apart from networking, entirely standard for an ML-like programming language. The system-step relation represents two operations: delivering sent message to the receiving socket's buffer and dropping not-yet-delivered sent messages. A detailed description of networking and

[4]See Jung et al. [2018] for details of how this can be done in Iris.

system step relations of AnerisLang can be found in the Appendix. The evaluation contexts of AnerisLang are entirely standard for a call-by-value ML-like programming language—we refer to the accompanying Coq development for further details. In order to instantiate Trillium with AnerisLang we need to define the $\mathit{StateInterp}$ and $\mathsf{ValidEvolution}$ predicates. Before doing so we explicate the notion of the *model* that we will work with in the AnerisLang instantiation.

Models. A *model* is a state transition system (STS), $\mathcal{M} = (A_{\mathcal{M}}, \rightharpoonup_{\mathcal{M}}, \iota_{\mathcal{M}})$ where $A_{\mathcal{M}}$ is a set, $\rightharpoonup_{\mathcal{M}} \subseteq A_{\mathcal{M}} \times A_{\mathcal{M}}$ is an arbitrary *transition* relation, and $\iota_{\mathcal{M}} \in A_{\mathcal{M}}$ is the initial state. We call elements of $A_{\mathcal{M}}$ (model) states and the relation $\rightharpoonup_{\mathcal{M}}$ the (step) relation of $\mathcal{M}$. We say the model $\mathcal{M}$ *takes a step* from δ to δ' and write $\delta \rightharpoonup_{\mathcal{M}} \delta'$ whenever δ and δ' are related by $\rightharpoonup_{\mathcal{M}}$.

When instantiating Trilliumwith AnerisLang, we work with an arbitrary but fixed model; in practice the user picks the model based on the verification task at hand. Moreover, we take the set of auxiliary states to be the set of states of the model and let the ValidEvolution relation be:

$$\mathsf{ValidEvolution}(\tau, \mu, \sigma, \delta) \triangleq \mathit{last}(\mu) = \delta \vee \mathit{last}(\mu) \rightharpoonup_{\mathcal{M}} \delta$$

Note how this relation does not restrict the program execution trace, nor the program state—it merely requires that the new model state is either the same as the last one or that it is related to it by a *single step* of the model's transition relation. Note further that this definition allows for stuttering on the model side; this is natural given that we wish to relate a detailed program execution to a more abstract model. Technically, this also means that the only proof rule that needs to mention the model is the rule ANERIS-WP-ATOMIC-TAKE-STEP that we will explain below; all other rules of the logic can ignore the existence of the model since we can simply pick the same model state as the last state when we have to show that there is a matching state in the model, and thus we get a conservative extension of Aneris.

The state interpretation predicate is defined as follows:

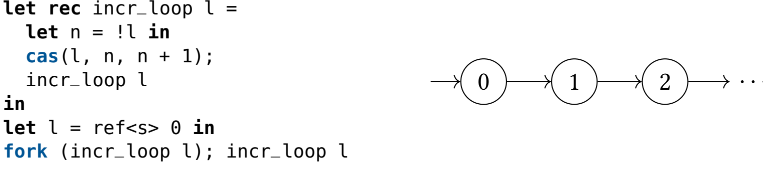
$$\mathit{StateInterp}(\tau, \mu) \triangleq \mathsf{eventSI}(\tau) * \mathsf{physSI}(\mathit{last}(\tau)) * \mathsf{Model}_{\bullet}(\mathit{last}(\mu))$$

Here, the physSI predicate associates the physical state, *i.e.*, heap, network, *etc.*, to Iris resources as explained earlier. The eventSI predicate associates resources to what we call events and allows to reason about them in the logic while reasoning about programs—in particular, we can track events for allocation of certain references, as well as send and receive events on certain network addresses; we will see examples of the use of allocation events later in this section, and examples of send and receive events in §4. The $\mathsf{Model}_{\bullet}$ predicate is defined in terms of Iris resources and allows the user to refer to the current state of the model. This is done in a fashion similar to how points-to propositions are used to refer to the physical state of the heap. The $\mathsf{Model}_{\bullet}$ predicate ("the authoritative part of the model") comes with a counterpart $\mathsf{Model}_{\circ}$ ("the fragmental part of the model").[5] The authoritative and fragmental parts of the model satisfy the following rule:

$$\mathsf{Model}_{\bullet}(\delta) * \mathsf{Model}_{\circ}(\delta') \vdash \delta = \delta'$$

Thus, one can use $\mathsf{Model}_{\circ}(\delta')$ to refer to the state of the model, which is tracked by and hidden inside the definition of weakest preconditions. We will briefly discuss a minimal example of how this is done after a quick detour into Iris resources and invariants, as well as the rule ANERIS-WP-ATOMIC-TAKE-STEP, which we will use in the minimal example that follows (and in other examples presented throughout the paper).

[5] These terminologies originate from the terminology of the so-called authoritative resource algebras in Iris; see Jung et al. [2018] for more details, including how the authoritative and exclusive resource algebras can be combined to define the authoritative and fragment parts of the model predicates.



```
let rec incr_loop l =
  let n = !l in
  cas(l, n, n + 1);
  incr_loop l
in
let l = ref<s> 0 in
fork (incr_loop l); incr_loop l
```


Fig. 1. A simple example and its corresponding model.

Iris Invariants. Iris's base logic features so-called invariants. In Iris the proposition $\boxed{P}^{\mathcal{N}}$ means that P is an invariant (intuitively, it should always hold) with the name $\mathcal{N}$. The base logic also features an update modality ${}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}}$ annotated with two masks $\mathcal{E}$ and $\mathcal{E}'$ which are sets of invariant names. We write ${\mid\!\Rrightarrow}_{\mathcal{E}}$ as a shorthand for ${}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}}$ and write $P \;{}^{\mathcal{E}}{\Rrightarrow\!\!*}^{\mathcal{E}'}\; Q$ as a shorthand for $P \mathrel{-\!\!*} {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}'} Q$. Furthermore, we write $\top$ for the mask that includes all invariant names. The proposition ${}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}'} P$ holds if P holds after updating resources and accessing/creating invariants under two conditions: (1) all invariants in $\mathcal{E}$ can be used to establish P, and (2) all invariants in $\mathcal{E}'$ should hold afterwards. The definition of weakest precondition $\mathsf{wp}_{\mathcal{E}}\; e \;\{\Phi\}$ is parameterized by a mask $\mathcal{E}$ indicating which invariants can be accessed during the proof. Our definition of weakest preconditions, just like weakest preconditions in Iris, allow invariants to be opened only for the duration of an atomic step.

Updating the State of the Model. The following inference rule allows the state of the model to be updated during execution of atomic operations.

ANERIS-WP-ATOMIC-TAKE-STEP

$$\frac{\delta \rightharpoonup_{\mathcal{M}} \delta' \qquad \mathrm{Atomic}(e) \qquad e \notin \mathit{Val}}{\mathsf{Model}_{\circ}(\delta) * \mathsf{wp}_{\mathcal{E}}\,\langle ip;\, e\rangle \,\{x.\, \mathsf{Model}_{\circ}(\delta') \mathrel{-\!\!*} \Phi(x)\} \vdash \mathsf{wp}_{\mathcal{E}}\,\langle ip;\, e\rangle\,\{x.\, \Phi(x)\}}$$

This rule essentially says that to prove a weakest precondition for the atomic operation e with a postcondition that follows when the model state is δ', it suffices to show that the current state δ can take a step to reach δ'. Note how this rule requires the program to take a single step: it needs to be atomic, so it takes *at most* a single step, and it is not a value, hence, according to the definition weakest preconditions, it *must* take a step.

A Minimal Example. Figure 1 shows a simple example (without distribution). The example is contrived, yet, it demonstrates our methodology for proving program refinements: we use invariants to relate the state of the program to the state of the model and subsequently, as we will see in the next section, use the adequacy theorem of Trillium to show the desired refinement relation.

The program first allocates a fresh location ℓ with initial value 0 and subsequently, in two concurrently running threads, enters an infinite loop that tries to increment ℓ using the compare-and-set (cas) command. Note how the allocation of the reference is marked with a label s, which allows us to refer to ℓ when we look at the execution trace. Such allocation events are useful to address the following issue. We would like to express a refinement relationship between the program and the model shown in the figure. Intuitively this refinement should say that the values stored in ℓ are the same as the states of the model. However, the location ℓ is not allocated at the beginning and hence we cannot refer to it formally—we do not even know what the concrete location will be, since our operational semantics permits it to be nondeterministically chosen. This is where our notion of events comes to the rescue! In Aneris we have a predicate $\mathsf{AllocEvs}_s(\mathit{evs})$, which intuitively means that evs is the list of allocation steps labeled by s that have taken place so far. In particular, we have the following rule:

$$\mathsf{AllocEvs}_s(\mathit{evs}) * \mathsf{eventSI}(\tau) \vdash \mathrm{TraceAllocs}_s(\tau) = \mathit{evs}$$

where $\mathrm{TraceAllocs}_s$ is a function that maps an execution trace to the list of its allocation steps labeled by s. To establish a refinement relation between the program and the model we use the following invariant, assuming that the code runs on a node with IP address ip:

$$incrInv \triangleq \boxed{\begin{array}{l}(\mathsf{AllocEvs}_s([\,]) * IncrLoc_\bullet(\mathsf{None}) * \mathsf{Model}_\circ(0)) \lor \\ \left(\begin{array}{l}\exists \ell, n.\ \mathsf{AllocEvs}_s([\,allocated(\ell)\,]) * \\ \quad IncrLoc_\bullet(\mathsf{Some}(\ell)) * \ell \mapsto_{ip} n * \mathsf{Model}_\circ(n)\end{array}\right)\end{array}}^{\mathcal{N}_{incr}}$$

This invariant states that either there has been no allocation with label s, in which case the model has value 0, or there has been exactly one location l allocated with label s, and its value is exactly the value corresponding to the state of the model. The predicate $IncrLoc_\bullet$ s and $IncrLoc_\circ$ are used for tracking whether the location has been allocated or not—we will see how below. These predicates satisfy the following rules:

INCR-LOC-CREATE
$$\Rrightarrow_{\mathcal{E}} IncrLoc_\bullet(\mathsf{None}) * IncrLoc_\circ(\mathsf{None})$$

INCR-LOC-AGREE
$$IncrLoc_\bullet(a) * IncrLoc_\circ(b) \vdash a = b$$

INCR-LOC-UPDATE
$$IncrLoc_\bullet(\mathsf{None}) * IncrLoc_\circ(\mathsf{None}) \Rrightarrow\!\!*_{\mathcal{E}}\ IncrLoc_\bullet(\mathsf{Some}(\ell)) * IncrLoc_\circ(\mathsf{Some}(\ell))$$

In these rules the update modality has an arbitrary mask as the rules do not interact with invariants and simply manipulate resources.

There are two points that need to be established about this example: (1) that we can prove the desired Hoare triple below, and (2) that proving this Hoare triple allows us to conclude that the program refines the model (discussed in the next section). For now, let us assume that at the beginning the invariant above is established and we have $IncrLoc_\circ(\mathsf{None})$. We then prove that the following weakest precondition holds (the postcondition is False as the program loops indefinitely):

$$\{incrInv * IncrLoc_\circ(\mathsf{None})\}\ \langle ip;\ \mathtt{let}\ l = \mathtt{ref}\langle s\rangle\ 0\ \mathtt{in}\ \mathtt{fork}\,(incr_loop\ l)\,;\ incr_loop\ l\rangle\ \{x.\ \mathsf{False}\}$$

To prove this, reasoning modularly, it suffices to show the following:

$$\{incrInv * IncrLoc_\circ(\mathsf{None})\}\ \langle ip;\ \mathtt{ref}\langle s\rangle\ 0\rangle\ \{x.\ \exists \ell.\ x = \ell * IncrLoc_\circ(\mathsf{Some}(\ell))\} \qquad \text{(incr-alloc)}$$

$$\forall \ell.\ \{incrInv * IncrLoc_\circ(\mathsf{Some}(\ell))\}\ \langle ip;\ incr_loop\ \ell\rangle\ \{x.\ \mathsf{False}\} \qquad \text{(incr-loop)}$$

We first show that (incr-alloc) holds. Since allocation is an atomic step we can access the invariant and since we have $IncrLoc_\circ(\mathsf{None})$, we can use rule INCR-LOC-AGREE to conclude that the right disjunct cannot hold. Hence we obtain $\mathsf{AllocEvs}_s([\,])$, $IncrLoc_\bullet(\mathsf{None})$, and $\mathsf{Model}_\circ(0)$. After the allocation operation we get a freshly allocated location ℓ for which we have $\ell \mapsto_{ip} 0$ and $\mathsf{AllocEvs}_s([\,allocated(\ell)\,])$. This is by an application of the following rule WP-ALLOC-WITH-LABEL in Aneris (for comparison, WP-ALLOC-WITHOUT-LABEL is for allocations without labels):

WP-ALLOC-WITH-LABEL
$$\{\mathsf{AllocEvs}_s(evs)\}\ \langle ip;\ \mathtt{ref}\langle s\rangle\ v\rangle\ \{x.\ \exists \ell, x = \ell * \ell \mapsto_{ip} v * \mathsf{AllocEvs}_s(evs +\!\!+ [\,allocated(\ell)\,])\}$$

WP-ALLOC-WITHOUT-LABEL
$$\{\mathsf{True}\}\ \langle ip;\ \mathtt{ref}\,v\rangle\ \{x.\ \exists \ell, x = \ell * \ell \mapsto_{ip} v\}$$

where $+\!\!+$ is the list append operation. At this point in the proof we use the rule INCR-LOC-UPDATE to obtain $IncrLoc_\bullet(\mathsf{Some}(\ell))$ and $IncrLoc_\circ(\mathsf{Some}(\ell))$ which allows us to reestablish the invariant by proving the right disjunct and prove the post condition.

By Löb induction (the proof principle used for reasoning about recursive functions), the proof of (incr-loop) boils down to showing that the Hoare triple (incr-loop) holds for the body of the loop. The body of the loop essentially consists of two operations: loading ℓ and the cas operation. Both of these operations are atomic and hence we can open invariants around them and since

we have $IncrLoc_{\circ}(\mathsf{Some}(\ell))$ we can exclude the left disjunct of the invariant. Furthermore, the load operation does not change the value of ℓ and it is trivial to reestablish the invariant after the load operation. The cas operation, on the other hand, can have two different outcomes: either it succeeds in updating the value of ℓ or it fails, and the value of ℓ remains unchanged. In the latter case the invariant can trivially be reestablished. In case the cas operation actually succeeds, the value of ℓ is incremented by 1 and hence, since cas is atomic, and not a value, the rule ANERIS-WP-ATOMIC-TAKE-STEP applies and we can update the state of the model in order to reestablish the invariant.

2.3 Adequacy of Trillium and Possibly-Infinite Traces.

The adequacy theorem of Trillium states that if a weakest precondition holds, then any history-sensitive refinement relation that can be shown to hold *inside Iris*, also holds between the program and the model *outside Iris* (in the meta-theory, *i.e.*, in Coq in practice). The adequacy theorem involves a technical condition: we require that the user-defined relation ValidEvolution is a *finitary relation*, *i.e.*, that the set $\{\delta \mid \mathsf{ValidEvolution}(\tau, \mu, \sigma, \delta)\}$ is finite, for any τ, μ, and σ. This condition is necessary as the underlying model of the base logic of Iris is step-indexed over the natural numbers (see the discussion after the explanation of the theorem).

Theorem 2.1 (Adequacy). *Let e be a program, σ be a state of the program, and Φ be an Iris predicate on values. Let δ be an auxiliary state. Let ξ be a binary relation relating execution traces to traces of the model and suppose that the user-defined relation* ValidEvolution *is a finitary relation. If the following is provable in Iris*

$$\models\!\!\Rrightarrow_{\top} SysWP * StateInterp((e;\sigma),\delta) * wp_{\top}\, e\,\{\Phi\} * AlwaysHolds(\xi)$$

then $Ref_{\xi}([(e,\sigma)]_{tr}, [\delta]_{tr})$ holds outside Iris. Here, $SysWP$ and $AlwaysHolds(\xi)$ are Iris predicates whereas Ref_{ξ} is a predicate in the meta-theory; they are defined as follows:

$$\begin{aligned}
SysWP \triangleq\ & \forall \tau, \mu, \sigma, \sigma'.\ state(last(\tau)) = \sigma \wedge \sigma \rightarrow_{\mathsf{sys}} \sigma' \Rightarrow \\
& StateInterp(\tau,\mu) \Rrightarrow\!\!*_{\top} \\
& \exists \delta.\ \mathsf{ValidEvolution}(\tau, \mu, update_{st}(\sigma', last(\tau)), \delta) * \\
& StateInterp(\tau ::_{tr} update_{st}(\sigma', last(\tau)), \mu ::_{tr} \delta)
\end{aligned}$$

$$\begin{aligned}
AlwaysHolds(\xi) \triangleq\ & \forall \tau, \mu.\ first(\tau) = (e;\sigma) * first(\mu) = \delta * \\
& \begin{pmatrix} \forall \tau', \mu', c', \delta'.\ \tau = \tau' ::_{tr} c' \wedge \mu = \mu' ::_{tr} \delta' \Rightarrow \\ \mathsf{ValidEvolution}(\tau', \mu', c', \delta') \wedge \xi(\tau', \mu') \end{pmatrix} * \\
& StateInterp(\tau,\mu) * \\
& \begin{pmatrix} \forall e_1, \ldots, e_n, \sigma.\ last(\tau) = (e_1, \ldots, e_n; \sigma) \Rightarrow \\ (\forall 1 \leq i \leq n.\ e_i \in Val \vee reducible(e_i, \sigma)) \wedge \\ (e_1 \in Val \Rightarrow \Phi(e_1)) \end{pmatrix} \\
& {}^{\top}\!\Rrightarrow\!\!*^{\emptyset}\ \xi(\tau,\mu)
\end{aligned}$$

Ref_{ξ} is defined as the greatest fixpoint (i.e., a coinductive definition) of the following equation:

$$\begin{aligned}
Ref_{\xi}(\tau,\mu) \ =\ & \xi(\tau,\mu) \wedge \forall c.\ last(\tau) \rightarrow_{\mathsf{tp}} c \Rightarrow \\
& \exists \delta.\ \mathsf{ValidEvolution}(\tau, \mu, c, \delta) \wedge Ref_{\xi}(\tau ::_{tr} c, \mu ::_{tr} \delta)
\end{aligned}$$

This theorem essentially says that we have a history-sensitive refinement relation between the execution trace and the auxiliary trace for the initial (singleton) traces, if we establish the following:

(1) $SysWP$: System steps preserve our representation of resources ($StateInterp$) and our refinement with the auxiliary trace (ValidEvolution).
(2) $AlwaysHolds(\xi)$: At any point during the execution of the program, *i.e.*, given any execution trace τ and any auxiliary trace μ that start in the initial configuration $(e; \sigma)$ and the initial auxiliary state δ, we can prove $\xi(\tau, \mu)$ assuming that the following holds:
   (a) If τ and μ are results of taking steps (*i.e.*, they are not just the singleton initial traces), then ξ holds for the traces up to before the very last step and the last step has been a valid step according to ValidEvolution,
   (b) The state interpretation holds for τ and μ, and
   (c) Non of the threads at the current execution point ($last(\tau)$) are stuck, and if the first thread (corresponding to e) has terminated to a value, then the postcondition Φ holds for it.

We will explain how this theorem is used below. The only caveat in this theorem is the side condition that ValidEvolution must be finitary. This condition is necessary because the underlying model of the base logic of Iris is step-indexed over the natural numbers, which means that existentially quantified predicates inside Iris do not correspond to existentially quantified predicates outside of Iris, unless the domain of the existential quantifier is finite (here the new auxiliary state is existentially quantified in the definition of weakest preconditions). The finiteness side condition is a standard technique to work around this limitation [Tassarotti et al. 2017]. Note that the condition does not restrict the properties we can transport along a refinement; in §4 and §5 we will see examples of how we can transfer liveness properties (fairness, termination, and eventual consistency) along refinements. One possible approach to avoid the finiteness condition is to use the recently proposed Transfinite Iris [Spies et al. 2021] as the base logic. However, it is not obvious that one can carry out our development in Transfinite Iris, since Transfinite Iris does not include all of the basic reasoning principles of standard Iris (in particular, Transfinite Iris lacks commutation rules for the later modality).

Refinements for Infinite Executions. Ref_{ξ} is defined by a coinductive definition and says that for any step taken by the program there is an auxiliary state by which we can extend the auxiliary trace such that the extended traces satisfy ξ. Therefore, given an infinite execution of the program we should be able to obtain an infinite auxiliary trace such that all corresponding finite prefixes are related. This is precisely what the following theorem states. In this theorem, as well as the rest of this paper, whenever we speak of predicates over a possibly-infinite execution/possibly-infinite trace, we mean predicates on pairs of traces: one finite trace, corresponding intuitively to the trace up to now, and a possibly-infinite trace, corresponding to the rest of the execution. For instance, the following coinductive predicate describes what we call a valid infinite execution (similar to $ValidExec$ for finite execution traces):

$$\begin{aligned} &ValidInfExec(\tau, \ddot{\tau}) \triangleq (\, ValidExec(\tau) \land \ddot{\tau} = nil) \lor \\ &\quad (\exists c, \ddot{\tau}'.\ ValidExec(\tau) \land last(\tau) \to_{\mathsf{tp}} c \land \ddot{\tau} = c :: \ddot{\tau}' \land ValidInfExec(\tau ::_{tr} c, \ddot{\tau}')) \end{aligned}$$

We define a refinement relation between possibly-infinite execution traces and auxiliary traces as a coinductive definition as follows:

$$\begin{aligned} &InfRef_{\xi}(\tau, \mu, \ddot{\tau}, \ddot{\mu}) \triangleq (\xi(\tau, \mu) \land \ddot{\tau} = nil \land \ddot{\mu} = nil) \lor \\ &\quad (\exists c, \delta, \ddot{\tau}', \ddot{\mu}'.\ \ddot{\tau} = c :: \ddot{\tau}' \land \ddot{\mu} = \delta :: \ddot{\mu}' \land \xi(\tau', \mu') \land InfRef_{\xi}(\tau ::_{tr} c, \mu ::_{tr} \delta, \ddot{\tau}', \ddot{\mu}')) \end{aligned}$$

This definition essentially states that all finite prefixes satisfy ξ.

Theorem 2.2 (Possibly-Infinite Refinement). *Let τ, μ be traces, and ξ be a relation such that $Ref_{\xi}(\tau,\mu)$ holds. Furthermore, let $\ddot{\tau}$ be a possibly-infinite trace such that $ValidInfExec(\tau,\ddot{\tau})$ holds. Then there exists a possibly-infinite auxiliary trace $\ddot{\mu}$ such that $InfRef_{Ref_{\xi}}(\tau,\mu,\ddot{\tau},\ddot{\mu})$ holds.*

Adequacy for AnerisLang: The Minimal Example Revisited. The adequacy theorem for Trillium instantiated with AnerisLang is a simple corollary of the adequacy theorem above and very close to it; hence we will elide it here. The only difference is that $SysWP$ is proven once and for all and hence not part of the adequacy theorem. Furthermore, the weakest precondition is required to hold only under the assumption that all network resources have been instantiated, *e.g.*, socket protocols for globally-fixed sockets, and that we know that there are initially no (allocation, send, or receive) events. This is because the network and event resources are initialized once and for all as part of the adequacy theorem of Aneris. Given this adequacy theorem for Aneris we can prove that the program and the model in Figure 1 are related using $Ref_{\xi_{incr}}$ where ξ_{incr} is the following relation:

$$\xi_{incr}(\tau,\mu) \triangleq (\mathrm{TraceAllocs}_s(\tau) = [] \land \forall i < length(\mu).\ get(i,\mu) = 0) \lor \left(\begin{array}{l} \exists \ell, n.\ \mathrm{TraceAllocs}_s(\tau) = [allocated(\ell)] \land \\ \quad Heap(ip, get(i, last(\tau)))(\ell) = n \land last(\mu) = n \land \\ \quad \left(\begin{array}{l}\forall i < length(\tau).\ \ell \in \mathrm{dom}(Heap(ip, get(i,\tau))) \Rightarrow \\ \quad \exists m \le n.\ Heap(ip, get(i,\tau))(\ell) = m \land get(i,\mu) = m\end{array}\right) \land \\ \quad (\forall m \le n.\ \exists i < length(\tau).\ \exists m \le n.\ Heap(ip, get(i,\tau)(\ell)) = m \land get(i,\mu) = m) \end{array}\right)$$

This relation says one of the following cases holds:

- No location with label s has been allocated and the model trace consists of 0's.
- The location ℓ is the only location allocated with label s and there is a number n such that:
  - The current values of the heap at ℓ and the model are exactly n.
  - Whenever, in the past, ℓ is allocated, *i.e.*, it is in the domain of the heap, both the value of ℓ as well as the value of the model are some $m \le n$.
  - For any value $m \le n$ there is a point in the past where the value of ℓ and the model are both exactly m.

In other words, it captures precisely the refinement relation would we expect.

To show that $Ref_{\xi_{incr}}$ holds for the initial (singleton) program and model traces we need to show that we can satisfy the requirements of the adequacy theorem. Note that the ValidEvolution relation in this case is trivially finitary: the new model state is either the same as the last state or it has been incremented by one. Hence, we only need to show that the weakest precondition and $AlwaysHolds(\xi_{incr})$ hold. The former is straightforward as we only need to establish the precondition of the Hoare triple that we discussed earlier. This can be easily done by allocating location tracking resources $IncrLoc_{\bullet}$(None) and $IncrLoc_{\circ}$(None) using the rule INCR-LOC-CREATE and allocating the invariant. Hence, we only need to show $AlwaysHolds(\xi_{incr})$. However, ξ_{incr} simply holds for the initial traces as the left disjunct trivially applies. Otherwise, we need to show $\xi_{incr}(\tau,\mu)$ such that $\tau = \tau' ::_{tr} c$ and $\mu = \mu' ::_{tr} \delta$. However, according to the definition of $AlwaysHolds$, we know $\xi_{incr}(\tau',\mu')$ and that $\delta = last(\mu) \lor \delta = last(\mu) + 1$ (because of ValidEvolution in $AlwaysHolds$). These facts, together with the fact that the invariant and the state interpretation imply that the value of ℓ is equal to δ allow us to prove that $\xi_{incr}(\tau,\mu)$.

3 REFINEMENT OF TLA+ SPECIFICATIONS

In the following, we show how to establish our refinement relation between implementations of two classical distributed algorithms, two-phase commit and single-decree Paxos (SDP), and their

TLA⁺ models. As simple corollaries of the refinement relation and our adequacy theorems we establish using the *same* modular specification (1) that clients are *safe*, *i.e.*, they do not crash, (2) a formal proof that the AnerisLang implementation correctly implements the protocol specification, and (3) correctness of the implementation by leveraging already-established correctness properties of the models. Both the TLA⁺ specification of transaction commit and the TLA⁺ specification of single-decree can also be found in the official TLA⁺-examples repository on GitHub.

Due to space constraints, we omit network-related Aneris resources and focus on the core parts of showing the refinement. Both the implementation, model, and refinement (with network resources) for the two-phase commit protocol can be found in the Appendix. The development follows the same methodology as for the Paxos algorithm which we describe below.

3.1 Single-Decree Paxos

The Paxos algorithm [Lamport 1998, 2001] 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 roles 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 fulfill their promise and accept its value. To satisfy the safety 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 grater 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 2. 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 $\mathcal{S} \in \mathscr{P}(\mathsf{PaxosMessage})$ and two maps $\mathcal{B} : \mathsf{Acceptor} \to \mathsf{Option}(\mathsf{Ballot})$ and $\mathcal{V} : \mathsf{Acceptor} \to \mathsf{Option}(\mathsf{Ballot} \times \mathsf{Value})$ 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

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

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

The SDP-Phase1a transition adds a $\mathsf{msg1a}(b)$ 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 $\mathsf{msg1a}(b)$ 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 $\mathsf{msg1b}(a, b, o)$ 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 $\mathsf{msg2a}(b, v)$ 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 such that the $\mathit{ShowsSafeAt}(\mathcal{S}, Q, b, v)$ 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

$$
\begin{aligned}
Q1bv(\mathcal{S}, Q, b) &\triangleq \{m \in \mathcal{S} \mid \exists a, v.\, m = \mathsf{msg1b}(a, b, \mathsf{Some}(v)) \land a \in Q\} \\
HavePromised(\mathcal{S}, Q, b) &\triangleq \forall a \in Q.\, \exists m \in \mathcal{S}, o.\, m = \mathsf{msg1b}(a, b, o) \\
IsMaxVote(\mathcal{S}, Q, b, v) &\triangleq \exists m_0 \in Q1bv(\mathcal{S}, Q, b), a_0, b_0.\, m = \mathsf{msg1b}(a_0, b, \mathsf{Some}(b_0, v)) \land \\
&\qquad \forall m' \in Q1bv(\mathcal{S}, Q, b). \\
&\qquad\quad \exists a', b', v'.\, m' = \mathsf{msg1b}(a', b, \mathsf{Some}(b', v')) \land b_0 \geq b' \\
ShowsSafeAt(\mathcal{S}, Q, b, v) &\triangleq HavePromised(\mathcal{S}, Q, b) \land \\
&\qquad (Q1bv(\mathcal{S}, Q, b) = \emptyset \lor IsMaxVote(\mathcal{S}, Q, b, v))
\end{aligned}
$$

SDP-Phase1a

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

SDP-Phase1b

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

SDP-Phase2a

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

SDP-Phase2b

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

Fig. 2. TLA+ specification of single-decree Paxos (SDP).

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 $\mathsf{msg2b}(a, b, v)$ message and updating its state to reflect this fact. A value v has been chosen when a quorum of acceptors have sent a $\mathsf{msg2b}(a, b, v)$ message for some ballot b:

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

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

Theorem 3.1 (Consistency, SDP). *Let* $\iota_{\mathsf{SDP}} = (\emptyset, \lambda_.\, None, \lambda_.\, None)$. *If* $\iota_{\mathsf{SDP}} \rightharpoonup^*_{\mathsf{SDP}} (\mathcal{S}, \mathcal{B}, \mathcal{V})$ *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 in AnerisLang 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 a 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 kind 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

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.
```

fact to all the learners. The learner implementation (see 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 $\{k|\mathsf{Proposer}| + p \mid k \in \mathbb{N}\}$ by keeping track of the last issued k in a local reference.

3.2 Consensus by Refinement

To show that the SDP implementation refines the SDP model we would instantiate the Aneris logic with the model but this model is *not* finitary and the adequacy theorem would not hold: the SDP-Phase1a and SDP-Phase2a transitions can be made for *any* ballot $b \in \mathsf{Ballot}$ and Ballot is a countably infinite set. However, the actual proposer implementation does not issue ballots non-deterministically as ballots are issued based on a local counter as described above. By adding this information at the model level we obtain a finitary model. Concretely, we lift the SDP model to $\widehat{\mathsf{SDP}}$ that in addition to the state of SDP also keeps a map of counters $C \in \mathsf{Proposer} \to \mathbb{N}$ as part of its state and restricts the issued ballots to be derived from the counter; the Appendix contains the full definition. Importantly, this model *is* finitary an refines the SDP model in the following sense:

Lemma 3.2. *Let* $\iota_{\widehat{\mathsf{SDP}}} = (\lambda_.\, 0, \iota_{\mathsf{SDP}})$. *If* $\iota_{\widehat{\mathsf{SDP}}} \rightharpoonup^*_{\widehat{\mathsf{SDP}}} (C, \delta)$ *then* $\iota_{\mathsf{SDP}} \rightharpoonup^*_{\mathsf{SDP}} \delta$

We instantiate the Aneris logic with the lifted model; the key part of the proof is to keep the $\mathsf{Model}_\circ(C, \delta)$ 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 adequacy theorem (Theorem 2.1) to be strong enough for proving our final correctness theorem (Corollary 3.3). 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 $Msgs_\bullet(\mathcal{S})$ and $Msgs_\circ(m)$ such that $Msgs_\bullet(\mathcal{S}) * Msgs_\circ(m) \vdash m \in \mathcal{S}$, (2) maps, *e.g.*, with predicates $MaxBal_\bullet(\mathcal{B})$ and $MaxBal_\circ(a, b)$ such that $MaxBal_\bullet(\mathcal{B}) * MaxBal_\circ(a, b) \vdash \mathcal{B}(a) = b$, and (3) ballots with predicates $pending(b)$ and $shot(b, v)$ such that $pending(b) * shot(b, v) \vdash \mathsf{False}$, $pending(b) * pending(b) \vdash \mathsf{False}$, $pending(b) \Rrightarrow_{\mathcal{E}} shot(b, v)$, and $shot(b, v_1) * shot(b, v_2) \vdash v_1 = v_2$. Equipped with these resource we can state the invariant:

$$I_{\mathsf{SDP}} \triangleq \boxed{\begin{array}{c}\exists C, \mathcal{S}, \mathcal{B}, \mathcal{V}.\, \mathsf{Model}_\circ(C, (\mathcal{S}, \mathcal{B}, \mathcal{V})) * Msgs_\bullet(\mathcal{S}) * MaxBal_\bullet(\mathcal{B}) * MaxVal_\bullet(\mathcal{V}) * \\ Ctr_\bullet(C) * BalCoh(\mathcal{S}) * MsgCoh(\mathcal{S})\end{array}}^{\mathcal{N}_{\mathsf{SDP}}}$$

The first part of the invariant ties the current state of the model to its logical *authoritative* counterparts which means that by owning a *fragmental* part you own a piece of the model: *e.g.*, by owning $MaxBal_\circ(a, b)$ 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 $MaxBal_\circ(a, b)$ and $MaxVal_\circ(a, o)$) and similarly for proposer p (through $Ctr_\circ(p, k)$). Similarly, by owning $Msgs_\circ(m)$ you may conclude that the message m has in fact been added to the set of messages in the model; this predicate we will transfer along 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 $shot(b, v)$ holds. This implies that by owning $pending(b)$ 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, the proof must open the invariant, use ANERIS-WP-ATOMIC-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:

$$\{I_{\mathsf{SDP}} * MaxBal_\circ(a, \mathsf{None}) * MaxBal_\circ(a, \mathsf{None}) * \ldots\}\ \langle ip;\ \mathtt{acceptor}\ L\ a\rangle\ \{\mathsf{False}\}$$

$$\{I_{\mathsf{SDP}} * Ctr_\circ(p, k) * pending(b) * \ldots\}\ \langle ip;\ \mathtt{proposer}\ A\ skt\ b\ v\rangle\ \{\mathsf{True}\}$$

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 compose these specifications to prove both safety and the refinement relation for an arbitrary distributed system consisting of n proposers, m acceptors, and k learners.

Given the refinement relation for the full system, we can state and prove that the consistency property holds for all executions of the system. Let

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

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

$$\xi_{\mathsf{SDP}}(\tau, \mu) \triangleq \exists \mathcal{S}.\, last(\mu) = (_, (\mathcal{S}, _, _)) \land Messages(last(\tau)) \sim \mathcal{S}$$

we combine the adequacy theorem (Theorem 2.1) with our lifting lemma (Lemma 3.2) and model correctness theorem (Theorem 3.1) to obtain the following corollary that *only* talks about the execution of the SDP implementation.

Corollary 3.3. *Let e be the distributed system obtained by composing n proposers, m acceptors, and k learners. For any T and σ, if $(e; \emptyset) \twoheadrightarrow^* (T; \sigma)$ and both $ChosenI(Messages(\sigma), v_1)$ and $ChosenI(Messages(\sigma), v_2)$ hold then $v_1 = v_2$.*

Verifying Functional Correctness of Clients. Recall from the Introduction that our instantiation of Trillium to AnerisLang is an extension of Aneris. This also means that the original safety adequacy theorem of the Aneris logic still holds: by proving a Hoare triple about a program, the program is safe, *i.e.* it does not crash. 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. By proving a suitable Hoare triple for the client program, we may compose a distributed system containing the client together with proposers, acceptors, and learner nodes and derive a Hoare triple for the full system. Using this single specification, we can *both* show a consistency theorem like Corollary 3.3 and prove that the distributed system does not crash. See the accompanying Coq development for more details.

4 SPECIFICATION AND VERIFICATION OF CRDTS USING REFINEMENT

According to the CAP theorem [Brewer 2000] no distributed system can, simultaneously, satisfy all the three desired properties of distributed systems: consistency (all replicas always agree), availability (responsiveness), and partition tolerance (can function even if some nodes have crashed/disconnected). Hence, different distributed systems choose to sacrifice (part of) one of these three properties. The table on the right shows these properties for the examples presented in this paper. Conflict-free replicated data types [Shapiro et al. 2011] weaken the consistency of the system to so-called *eventual consistency* [Vogels 2009], which, loosely speaking, states that all replicas are guaranteed to be consistent once they have received the same set of updates from other replicas.

Example	Consistency	Availability	Partition Tolerance
Two-Phase Commit	✓	✓	
Paxos	✓		✓
CRDT	Eventual Consistency	✓	✓

In this section we use Trillium to reason about a CRDT called G-Counter (a grow-only replicated counter). Despite their simplicity, G-Counters illustrate subtle and salient aspects of specification and verification of eventual consistency of CRDTs when *(a)* it is done fully formally *(b)* for an actual implementation including replicas' intercommunication, *(c)* along with specifying and proving node-local functional correctness within the same formal setting (in Aneris and Coq).

Implementation. The implementation of the G-Counter in Figure 3 consists of the following:

- The `install` method is used to initialize instances of G-Counter on different replicas and returns two methods: `query`, to read the value, and `incr`, to increment it.
- The `broadcast` loop, forked by `install`, repeatedly sends the local state to other replicas.
- The `apply` loop, also forked by `install`, repeatedly updates the local state based on the states of the other replicas it receives over the network.

The state of G-Counter replicas is a vector (an array), one element for each replica (including themselves), with the jth element of the vector tracking the number of increments performed on the jth replica. Note how the `incr` method on the ith replica increments the ith element of the vector, and the `query` function returns the sum of the vector. The `apply` method updates the local state by taking, for each replica, the maximum value of its current state and the value it has received from the network—the `vect_join_max` function computes point-wise maximum of two vectors.

```
let install addrlst i s =
  let n = List.length addrlst in
  let m = ref<s> (vect_mk n 0) in
  let sh = socket () in
  socketbind sh (List.nth addrlst i);
  fork (apply m sh);
  fork (broadcast m sh addrlst i);
  (query m, incr m i)


let rec incr m i () =
  let t = !m in
  if cas m t (vect_inc t i)
  then ()
  else incr m i ()

let query m () = vect_sum !m
```

```
let rec perform_merge m m2 =
  let t = !m in
  if cas m t (vect_join_max t m2) then ()
  else perform_merge m m2

let apply m sh =
  let rec loop () =
    let (b, _) = receivefrom sh in
    let m2 = vect_deserialize b in
    perform_merge m m2; loop ()
  in loop ()

let broadcast m sh nodes i =
  let rec loop () =
    let msg = vect_serialize !m in
    send_to_all sh msg nodes i; loop ()
  in loop ()
```

Fig. 3. Implementation of a global counter.

Note the inherent node-local concurrency in this implementation; the `broadcast` and `apply` methods running concurrently alongside the client code which invokes increment and query methods. Hence, in this example we use advanced features of Aneris, *e.g.*, support for node-local concurrency. The idea of eventual consistency for G-Counters, which we will make formal later, is that if at some point no increment operation takes place on any replica, assuming some fairness properties about the network and scheduling, the states of all replicas will converge.

Functional Correctness. Unsurprisingly, the *node-local guarantees* that clients can get for querying and incrementing are much weaker than for two-phase commit or Paxos. In the absence of coordination, the G-Counter merely enforces that each client always observes the effect of its calls to increment, but the precise value of the counter cannot be known; we only know that it is monotonically increasing. Figure 4 shows the formal specifications for `incr` and `query` that we have proved and used to prove both safety and eventual consistency of CRDTs and their clients. [6] In the specs for both methods, the local state of the ith replica is represented by an *abstract predicate* $\mathsf{gcounter}(i, k)$ where k is an under approximation of its current value (the sum of all elements of the vector).

QuerySpec

$$\{\mathsf{gcounter}(i,k)\}\quad \langle ip_i;\ \mathtt{query}()\rangle \quad \{m.\ k \le m * \mathsf{gcounter}(i,m)\}$$

IncrSpec

$$\{\mathsf{gcounter}(i,k)\}\quad \langle ip_i;\ \mathtt{incr}()\rangle \quad \{().\ \exists m.\, k < m * \mathsf{gcounter}(i,m)\}$$

Fig. 4. G-Counter query and increment node-local specification.

4.1 Specifying and Proving Eventual Consistency by Refinement

In this section we show that G-Counter has the eventual consistency property. That is, any execution trace that is *network-fair*, defined below, and has a *stability point*, a point after which there is no increment, also has a *convergence point*, a point after which all replicas have the same local state.

[6] We omit the spec for `install` whose postcondition is straightforward (it returns a pair of methods (qr, ic) that satisfy the specifications QuerySpec and IncrSpec respectively), but whose precondition contains network-specific initialization conditions which are not relevant here, *e.g.* the address List.nth addrlst i being a pair (ip, p) where p a free port.

The high-level idea of our proof is as follows: We consider a simple model for G-Counter. We define eventual consistency (stability point implies convergence point) for both possibly-infinite execution traces and possibly-infinite model traces. We show that any possibly-infinite trace of the model that is *model-fair*, defined below, is eventually consistent. We show that given a possibly-infinite execution trace and its corresponding model trace, if the model trace is eventually consistent, then so is the execution trace. We prove that any possibly-infinite model trace corresponding to a possibly-infinite execution trace that satisfies network fairness properties is model-fair.

Model and Model Fairness. The state of the model we take for G-Counter with n replicas is simply a vector (of length n) of vectors (of length n), *i.e.* a square matrix. That is, for each replica we take a vector representing its local state. As expected, the initial state for our model is a square matrix where all elements are 0. We write ι_n^{GC} for this initial state where n is the number of G-Counter replicas. The model STS has two kids of transitions (Figure 5) corresponding to the two state-changing operations on G-Counter: incrementing and merging a message received from the network. The GC-IncrStep transition updates the state δ such that the ith element of the ith vector, $\delta_{i,i}$ is incremented. This is precisely what happens in the program during the increment operation. The GC-ApplyStep transition, on the other hand, updates the ith vector to be the result of merging (point-wise maximum) of the ith vector with some vector $\vec{v}$ which is, point-wise, less than ($\sqsubseteq$) the vector for some other (j^{th}) replica. The idea is that the vector being merged corresponds to the state of jth replica in the past—the jth replica could have been incremented after its state was sent over the network and before getting merged.

GC-IncrStep
$$\frac{}{\delta \rightarrow_{\mathsf{GC}} \delta[i \mapsto \delta_i[i \mapsto \delta_{i,i} + 1]]}$$

GC-ApplyStep
$$\frac{\vec{v} \sqsubseteq \delta_j}{\delta \rightarrow_{\mathsf{GC}} \delta[i \mapsto \delta_i \sqcup \vec{v}]}$$

Fig. 5. Transition relation for the global counter model.

In order to support simpler and more compact writing we introduce the following notation. Given a finite trace t and a possibly-infinite trace $\ddot{t}$, we write $Unroll_n(t, \ddot{t})$ for the finite trace obtained by taking the first n elements of $\ddot{t}$ (if there are n elements, otherwise as many as available) and appending them on t. As an example, we have $Unroll_1(t, a :: \ddot{t}) = t ::_{tr} a$. We write $Drop_n$ for dropping the first n elements of a possibly-infinite trace. For example, we have $Drop_1(a :: \ddot{t}) = \ddot{t}$.

We define fairness for a model trace and a possibly-infinite model trace as follows:

$$ModelFair(\mu, \ddot{\mu}) \triangleq \forall i, j, k.\ \exists k'.\ last(Unroll_k(\mu, \ddot{\mu}))_i \sqsubseteq last(Unroll_{k'}(\mu, \ddot{\mu}))_j$$

This definition simply states that for any replicas i and j, for any number of steps k, there is a k' such the vector for replica j at step k' is greater than or equal to the vector for replica i at step k. In other words, it is *always* the case that the current state of ith replica is *eventually* merged into the jth replica.[7] We define eventual consistency, stability point, and convergence point as follows:

$$ModelStab_{\vec{v}}(\mu, \ddot{\mu}) \triangleq \exists k.\ \forall k'.\ \forall i.\ last(Unroll_{k+k'}(\mu, \ddot{\mu}))_{i,i} = \vec{v}_i$$
$$ModelConv_{\vec{v}}(\mu, \ddot{\mu}) \triangleq \exists k.\ \forall k'.\ \forall i.\ last(Unroll_{k+k'}(\mu, \ddot{\mu}))_i = \vec{v}$$
$$ModelEvCons_{(}\mu, \ddot{\mu}) \triangleq \forall \vec{v}.\ ModelStab_{\vec{v}}(\mu, \ddot{\mu}) \Rightarrow ModelConv_{\vec{v}}(\mu, \ddot{\mu})$$

The predicate $ModelStab_{\vec{v}}$ states that there exists a point k after which the diagonal of the state is exactly $\vec{v}$. Similarly, the predicate $ModelConv_{\vec{v}}$ states that there is a point k after which all local states are exactly $\vec{v}$.

[7] In the accompanying Coq formalization we mix representation of trace properties using the $Unroll$ and temporal logic-style modalities, *always* and *eventually* which we define for our notion of traces.

Theorem 4.1 (Model Eventual Consistency). *For all μ and $\ddot{\mu}$, if $ModelFair(\mu, \ddot{\mu})$ then $ModelEvCons_{(}\mu, \ddot{\mu})$.*

Closed System, Network Fairness, and Eventual Consistency. For the rest of this section we assume that we have a closed system consisting of a number of nodes where each node runs a client of G-Counters after initializing a local instance. That is, each node in the system runs a program

```
let (qr, ic) = install addrlst i i in client_i qr ic
```

where `addrlist` is the list of socket addresses of all replicas and $\texttt{client}_i$ is some arbitrary code that runs on the ith node as a client of the G-Counter—note how the label of the allocated location for the state of the ith node is i. For clients we only assume that they satisfy a Hoare triple where the precondition requires the query and the increment functions satisfy their specs given in Figure 4. We write c_n^{GC} for the initial configuration of the closed system G-Counters of n replicas.

We now proceed to show that any closed system of G-Counters has the eventual consistency property. As expected from earlier high-level informal proofs [Shapiro et al. 2011], this is based on a fairness assumption of the network. Since we are considering a concrete implementation here, we additionally assume some liveness properties of the implementation (including fairness of schedulers on the different nodes in the system), *e.g.*, that a message is eventually received if the network has not dropped it. The assumptions are as follows:

$$
\begin{aligned}
NetFairSend(\tau, \ddot{\tau}) &\triangleq \forall i, j, n.\ \exists k.\ n \leq length(\mathrm{TraceSends}_{i,j}(Unroll_k(\tau, \ddot{\tau})))\\
NetFairRec(\tau, \ddot{\tau}) &\triangleq \forall i, n.\ \exists k.\ n \leq length(\mathrm{TraceRecs}_i(Unroll_k(\tau, \ddot{\tau})))\\
NetFairDel(\tau, \ddot{\tau}) &\triangleq \forall i, j, sev, k.\ sev \in \mathrm{TraceSends}_{i,j}(Unroll_k(\tau, \ddot{\tau})) \Rightarrow\\
&\quad \exists k', sev', rev.\ \mathrm{same_or_happens_after}(sev', sev) \wedge msg(sev') = msg(rev) \wedge\\
&\qquad sev' \in \mathrm{TraceSends}_{i,j}(Unroll_{k+k'}(\tau, \ddot{\tau})) \wedge rev \in \mathrm{TraceRecs}_j(Unroll_{k+k'}(\tau, \ddot{\tau}))\\
NetFair(\tau, \ddot{\tau}) &\triangleq NetFairSend(\tau, \ddot{\tau}) \wedge NetFairRec(\tau, \ddot{\tau}) \wedge NetFairDel(\tau, \ddot{\tau})
\end{aligned}
$$

where $\mathrm{TraceSends}_{i,j}$ is the list of all send events from i^{th} replica to j^{th} replica and $\mathrm{TraceRecs}_i$ is the list of all receive events on i^{th} replica. The fairness criterion $NetFairDel$ simply says that for any send event sev from i^{th} replica to j^{th} replica, there is a send event sev' also sent from i^{th} replica to j^{th} replica that is received by the jth node. Moreover, sev' is either the same as sev or it is sent after it. Note how this definition allows for messages to be dropped but essentially only requires that *always eventually* a message is delivered from any node to any other node.

We define eventual consistency, stability point, and convergence point for a closed system just as we defined them for the model; instead of the state of the model we refer to the values stored on the heap of each replica. However, this is not immediately expressible as at beginning of execution the memory is not allocated. Hence, we follow an approach similar to example in Figure 1 in §2. The eventual consistency theorem that we prove about our implementation is as follows:

Theorem 4.2 (Eventual Consistency). *Let $\ddot{\tau}$ be a possibly-infinite trace such that $ValidInfExec(\left[c_n^{GC}\right]_{tr}, \ddot{\tau})$, and $NetFair(\left[c_n^{GC}\right]_{tr}, \ddot{\tau})$ hold. Then there exist $k \in \mathbb{N}$, $\ddot{\mu}$, and n locations $\ell_1, \ldots, \ell_n$ such that*

$$
\begin{aligned}
&\forall k'.\ \mathsf{GcMainRel}\left((\ell_1, \ldots, \ell_n), Unroll_{k+k'}\left(\left[c_n^{GC}\right]_{tr}, \ddot{\tau}\right), Unroll_{k+k'}\left(\left[\iota_n^{GC}\right]_{tr}, \ddot{\mu}\right)\right) \text{ and}\\
&EvCons_{\ell_1,\ldots,\ell_n}\left(Unroll_k\left(\left[c_n^{GC}\right]_{tr}, \ddot{\tau}\right), Drop_k(\ddot{\tau})\right)
\end{aligned}
$$

This theorem essentially says that there eventually is a point where all replicas have allocated their locations (part of GcMainRel as explained below) and that as of that point if there is a stability point, there must also be a convergence point. Intuitively, the relation GcMainRel holds when:

```
let rec yes b n = if cas b 1 0 then n := !n-1;
  if n > 0 then yes b n

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

let start k = let b = ref 0 in
  (yes b (ref k) || no b (ref k))
```

Fig. 6. The Yes and No threads.

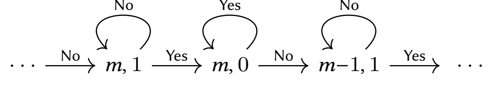


Fig. 7. The model $\mathcal{F}_{\mathsf{YN}}$.

(1) For any i, ℓ_i is the only location allocated with label i in the system (uniquely determined) and no messages are sent or received by node i before this location is allocated.
(2) For any vector $\vec{v}$ sent from node i to node j, $\vec{v}$ is point-wise greater than or equal to *the vector stored on the heap* at node i at the time of all previous sent messages from node i to node j.
(3) At all times, the vector stored on the heap of node i is point-wise greater than or equal to all vectors received by node i, *except possibly for the very last received message.*
(4) At all times, the vectors stored on the heap and the model agree.

Note the subtlety of condition (2) as it is capturing precisely the interaction between the program scheduler and the network steps: the vector on heap at the time of a send operation might be larger than the vector being sent as increment operations can take place in between the reading and the sending operations. Also, for condition (3), since the program receives messages in a loop and then subsequently merges them it might be that the last vector received is not yet merged.

Proof Idea of Theorem 4.2. Note how GcMainRel together with $NetFair$ implies $ModelFair$. Assume that $\vec{v}$ is the current state of ith replica (both in the model and the heap as they agree). At some point in the future, there is a vector $\vec{v'}$ sent from i^{th} replica to j^{th} replica that is received, such that $\vec{v} \sqsubseteq \vec{v'}$. On the other hand, the jth replica keeps receiving messages and once it gets a message after $\vec{v'}$, its state is guaranteed to be greater than or equal to $\vec{v'}$ and therefore also greater than or equal to $\vec{v}$. Moreover, since the vectors on the heap and the vectors in the model correspond at all times, the stability point and convergence point of the program and the model also correspond.

The Invariant. The invariant that we use for establishing the refinement relation between G-Counters and their models is as follows:

$$\boxed{\exists locs, \delta.\ \mathsf{Model}_\circ(\delta) * HeapRel(locs, \delta) * SentRel(locs, \delta) * RecRel(locs, \delta)}^{\mathcal{N}_{GC}}$$

Here the predicate $HeapRel$ states, as in the example in Figure 1, that either the ith replica has allocated its location and it stores exactly δ_i or δ_i is all 0's. This invariant captures the relation GcMainRel above using allocation, send, and receive events. Here, $locs$ is a list of optional locations (instead of a list of locations in GcMainRel). The $SentRel$ and $RecRel$ predicates, respectively, use send and receive events to state the criteria (2) and (3) in the explanation above for GcMainRel. In order to maintain the invariant above, we use the rule ANERIS-WP-ATOMIC-TAKE-STEP to update the state of the model whenever the `cas` operations in the `perform_merge` or `incr` methods succeed.

5 FAIR TERMINATION OF CONCURRENT PROGRAMS

We now consider an instantiation of Trillium to the HeapLang language, a concurrent higher-order language without network capabilities. We explain how instantiating Trillium with a suitable model allows proving fair termination of concurrent programs.

In a concurrent setting, the generally relevant notion of termination is *fair termination*, as most concurrent programs only terminate if the scheduler is fair. For example, the program presented in Figure 6, where two threads `yes` and `no` flip back and forth a shared Boolean `b`, does intuitively

terminate. However, it does not terminate if, after some point, only one thread is ever scheduled; this should not happen under a reasonable scheduler. By definition, fair termination of a program means that all its *fair traces* are finite. An execution trace $(\mathsf{tp}_1, \sigma_1) \xrightarrow{\mathsf{tid}_1}_{\mathsf{tp}} (\mathsf{tp}_2) \xrightarrow{\mathsf{tid}_2}_{\mathsf{tp}} \cdots$, whose transitions are labeled with the indices tid of the threads which take steps, is *fair* if it is finite, or if every reducible thread *eventually* takes a step.

Fair termination is a liveness property, and hence we cannot prove it directly in a step-indexed logic such as Iris (as discussed in [Spies et al. 2021; Tassarotti et al. 2017]). Our solution is to prove a *reduction* from the fair termination of an abstract model $\mathcal{F}$ to that of the program e:

$$\mathcal{F} \text{ is fairly terminating } \wedge \ e \text{ refines } \delta \quad \Longrightarrow \quad e \text{ is fairly terminating} \tag{1}$$

where $\delta \in \mathcal{F}$ and the fact that e refines δ is proved in Trillium. To express fairness, we use an instantiation of Trillium where models are labeled transition systems. Thus the model $\mathcal{F}$ above should be a *fairness model*: an STS labeled with names called *roles* which play the same role as thread ids in the definition of *fair traces* of $\mathcal{F}$. Each state $\delta \in \mathcal{F}$ has a finite set of enabled roles. For the example above, we define in Figure 7 a model $\mathcal{F}_{\text{YN}}$ with two roles, Yes and No corresponding to the two threads, and whose states are pairs $(m, b) \in \mathbb{N} \times \mathbb{B}$ which represent respectively the value of m and of b. Intuitively, the states of $\mathcal{F}_{\text{YN}}$ summarize the states of the program; note that if, initially, $n = m = k$ and $b = 1$, then $n = m + b$. Loops in $\mathcal{F}_{\text{YN}}$ represent failed **cas** operations and rightward arrows successful ones. This model is fairly terminating: at each state, one of the two roles decreases the state ordered lexicographically.

We need the refinement relation between the program e and the state δ of the fairness model $\mathcal{F}$ to induce a relation $\lesssim$ on their traces which entails (1), since fair termination is a trace property. The relation $\lesssim$ is complicated, as it needs to maintain an evolving mapping between thread ids and roles and to ensure finiteness of stuttering. We define it as the composition of two simpler relations $\lesssim_{\mathsf{f}}$ and $\lesssim_{\mathsf{s}}$ on traces so that $t_e \lesssim t_m$ iff there exists a trace t of an intermediate labeled STS $\mathsf{Live}(\mathcal{F})$ such that those two refinements hold:

$$t_e \underset{\text{finite}}{\overset{\text{fair}}{\rightleftarrows}}_{\lesssim_{\mathsf{f}}} t \underset{\text{finite}}{\overset{\text{fair}}{\rightleftarrows}}_{\lesssim_{\mathsf{s}}} t_m \tag{2}$$

The first relation $\lesssim_{\mathsf{f}}$ is defined as relation InfRef_{ξ} for a certain fixed ξ, and $\mathsf{Live}(\mathcal{F})$ designed to ensures that $\lesssim_{\mathsf{f}}$ corresponds to a fairness-preserving termination-preserving refinement. A state of $\mathsf{Live}(\mathcal{F})$ is a triple (δ, F, T) where $\delta \in \mathcal{F}$, F associates a natural number $F(\rho)$ to each role ρ enabled in δ which is called its *fuel*, and T associates each role with a thread id of the program. The idea is that a stutter step of thread tid decreases the fuels of all the roles associated to it according to T, and that a tid-step in the program which corresponds to a ρ step in $\mathcal{F}$, with $F(\rho) = \mathsf{tid}$, can *increase* the fuel of ρ but must *decrease* the fuel of all the other roles $\rho' \in T^{-1}(\mathsf{tid})$. This decreasing-fuel discipline ensures that there exists a computable function which extracts a trace t_m in $\mathcal{F}$ from a trace t of $\mathsf{Live}(\mathcal{F})$ by ignoring stuttering steps. The relation $\lesssim_{\mathsf{s}}$ is the graph of this function.

The correctness of this construction is represented by the gray arrows in (2): for example, the arrow from t_e to t means that if $t_e \lesssim_{\mathsf{f}} t$ and if t_e is fair, then t is fair. Thus if we assume that t_e is fair, then t_m is fair as well. If $\mathcal{F}$ is fairly terminating, we then get that t_m is finite, and therefore t_e is finite.

Coming back to our example, since $\mathcal{F}_{\text{YN}}$ is fairly terminating, it only remains to establish the refinement $t_e \lesssim_{\mathsf{f}} t$, which we do by proving a weakest precondition for the program e in Trillium instantiated with the model $\mathcal{M} := \mathsf{Live}(\mathcal{F})$. We use a slight variation of Trillium where the weakest precondition is parameterized by the current thread id (allowing to match thread ids to roles). We make use of ghost resources corresponding to the states of $\mathsf{Live}(\mathcal{M})$: in particular, $\rho \mapsto_F f$ means

that role ρ has fuel f, and $\mathsf{tid} \mapsto_T R$ means that the thread tid is associated to the set R of roles. Finally, $\mathsf{Model}_\circ(\delta)$ states that the current state of the underlying fairness model $\mathcal{F}$ is δ. Because fuel needs to decrease at each step, every program step of thread tid needs to be justified by owning the predicate $\mathsf{tid} \mapsto_T R$ (with $R \neq \emptyset$) and $\rho \mapsto_F f_\rho + 1$ for each $\rho \in R$; the $+1$ is consumed when tid takes a step. We can specialize the adequacy theorem of Trillium and use (2) to get:

Theorem 5.1. *Given a program e, a finitely branching fairness model $\mathcal{F}$, a state $\delta_0 \in \mathcal{F}$, if*

$$\mathsf{Model}_\circ\,\delta_0 \;-\!\!*\; 0 \mapsto_T R \;-\!\!*\; \mathop{\bigstar}_{\rho \in R} \rho \mapsto_F f_{init} \;-\!\!*\; {}^{\top}\!\!\Rrightarrow^{\top} wp_{\top}\, e \,@\, 0 \left\{0 \mapsto_T \emptyset\right\}$$

holds in Iris, and if $\mathcal{F}$ is fairly terminating, then e is fairly terminating. (0 *above is the initial thread id).*

We remark that the hypothesis that $\mathcal{F}$ is fairly terminating can be proved without quantifying over all fair traces: there is a simple criterion based on a well-founded order which can be checked locally by considering transitions individually.

A technical inconvenience is that threads need to have at least one role to take a step, but must have none when they end. In turn, this means that the last step of tid must take a step to a state in the model where its roles are not enabled. For our example, this leads to adding two Booleans ye and ne to the states, where $ye = 0$ means Yes has finished, and $ne = 0$ means No has.

Returning to our example we use the theorem above to prove that the program in Figure 6 is fairly terminating by establishing the weakest precondition, with $R := \{\mathsf{Yes}, \mathsf{No}\}$ and $\mathrm{f}_{init} := 30$. The proof of the weakest precondition is fairly simple and follows the methodology explained for the minimal example in §2. See the Appendix for complete definitions.

The approach presented here is similar in spirit to the one in the work of Tassarotti et al. [2017], but generalizes it from reasoning about fairness and termination-preserving refinement for a compiler for session-typed programs to reasoning about refinement of general concurrent programs wrt. abstract models. As far as we understand, the expressive power of the logics is roughly similar. The main difference is that Tassarotti et al. [2017] had to extend Iris with *linear* predicates, and to modify the definition of resource algebras 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 the weakening of $\mathsf{tid} \mapsto_T R_1 \cup R_2$ to $\mathsf{tid} \mapsto_T R_1$, a limited form of linearity.

6 RELATED WORK

We now discuss some further related work not already discussed in the paper.

Refinement-Based Verification of Distributed Systems. There is a lot of work on verification of high-level models of distributed systems, but here we focus on works that, as ours, aim at proving that concrete implementations refine abstract models. The most closely related works are IronFleet [Hawblitzel et al. 2017] and Igloo [Sprenger et al. 2020]. IronFleet uses the Dafny verifier to verify the implementation of a system and encode the relation to the STS being refined in preconditions and postconditions of programs. IronFleet does not support node-local concurrency and hence would not be applicable to our CRDT example. IronFleet uses a pen-and-paper argument for proving liveness of simple programs (programs that consist of a simple event loop which calls event handlers that are terminating), which does not scale to proving eventual consistency of our CRDT example. Igloo proves only safety properties about programs and not liveness properties like eventual consistency. Igloo starts with a high-level STS which is refined (possibly in multiple steps) to a more low-level STS for each node of the system. These 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]. The program (each node) is then verified against this generated

specification. Hence, 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 (history-sensitive) refinement relation to be specified and established between the program and the model. In contrast to both IronFleet and Igloo our verification approach is foundational: the operational semantics of the distributed programming language, the abstract models, and the model of the program logic 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.

Non-Refinement-Based Verification of Distributed Systems. Woos et al. [2016] verify the Raft consensus protocol [Ongaro and Ousterhout 2014] in the Verdi framework [Wilcox et al. 2015] for implementing and verifying distributed systems in Coq. In Verdi, the programmer provides a specification, implementation, and proof of a distributed system under an idealized network model in a high-level language. The application is automatically transformed into one that handles faults via verified system transformers: this makes vertical composition difficult for clients and the high-level language does not include features such as node-local concurrency. The Disel framework [Sergey et al. 2018] also allows users to implement distributed systems using a domain specific language and verify them using a Hoare-style program logic in Coq; the work includes a case study on two-phase commit. Disel struggles with node-local reasoning as the use of internal mutable state in nodes must be exposed in the high-level system protocol and state changes are tied to sending and receiving messages.

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; Kragl et al. 2020; 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.

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.

CRDTs. Zeller et al. [2014] present an Isabelle/HOL framework for verifying state-based CRDTs, including verifying that a CRDT implementation refines its specification. Unlike in our work, CRDT implementations are defined at a high level of abstraction using state-transition systems. Zeller et al. [2014] do not reason about inter-replica communication. Gomes et al. [2017] present the first mechanized proof of eventual consistency of operation-based CRDTs but do not consider network communications as part of the program. Moreover, unlike our work on a state-based CRDT, Gomes et al. [2017] do not consider functional correctness. Nair et al. [2020] present proof rules to reason about functional correctness of several state-based CRDTs that have richer safety guarantees than the CRDT we have studied because some operations of those CRDTs require coordination between replicas. However, they show safety and eventual consistency based on an abstract operational semantics which ignores inter-replica communication and node-local concurrency. Liang and Feng [2021] propose an approach to verify implementation of operation-based CRDTs where they show both functional correctness and strong eventual consistency within the same theoretical framework. They use a rely-guarantee-style program logic to reason about client programs, but do so at a higher "algorithmic" level of abstraction than our work, ignoring inter-replica communication. Furthermore, Liang and Feng [2021] do not mechanize their work in a

proof assistant. On the other hand, they consider many more examples of CRDTs than we do. Here, we have just focused on a single example, to illustrate how Trillium may be used to reason about CRDTs. In future work, it would be interesting to apply Trillium to other, more complex examples as well.

Fair Termination of Concurrent Programs. We have already discussed the most closely related work on fair termination via termination-preserving refinement in §5. 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.

7 CONCLUSION

We have introduced Trillium, a mechanized generic program logic that unifies Hoare-style reasoning with local reasoning about history-sensitive refinement relations among execution traces and traces of a model. We have shown how to use an instantiation of Trillium to a distributed higher-order concurrent imperative programming language to give modular proofs of correctness of concrete implementations of two-phase commit and single-decree Paxos by showing that they refine their abstract TLA+ specifications. Moreover, we have shown how our notion of refinement can be used to reason about liveness properties such as strong eventual consistency of a concrete implementation of a CRDT and fair termination of concurrent programs.

ACKNOWLEDGMENTS

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. During parts of this project Amin Timany was a postdoctoral fellow of the Flemish research fund (FWO).

REFERENCES

Robert Beers. 2008. Pre-RTL formal verification: an intel experience. In *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*. 806–811. https://doi.org/10.1145/1391469.1391675

Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*, Gil Neiger (Ed.). ACM, 7. https://doi.org/10.1145/343477.343502

R. M. Burstall and John Darlington. 1975. Some Transformations for Developing Recursive Programs. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). Association for Computing Machinery, New York, NY, USA, 465–472. https://doi.org/10.1145/800027.808470

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)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.), Vol. 9995. 119–136. https://doi.org/10.1007/978-3-319-48989-6_8

E.W. Dijkstra. 1970. *Notes on structured programming* (2nd ed. ed.). Technische Hogeschool Eindhoven.

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

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)*, Amal Ahmed (Ed.), Vol. 10801. Springer, 912–939. https://doi.org/10.1007/978-3-319-89884-1_32

Susan L. Gerhart. 1975. Correctness-Preserving Program Transformations. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, USA, January 1975*. 54–66. https://doi.org/10.1145/512976.512983

Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 109:1–109:28. https://doi.org/10.1145/3133933

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

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

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

C. A. R. Hoare. 1972. Proof of Correctness of Data Representations. *Acta Informatica* 1 (1972), 271–281. https://doi.org/10.1007/BF00289507

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, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 256–269. https://doi.org/10.1145/2951913.2951943

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, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

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, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. 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*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. 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, POPL 2017, Paris, France, January 18-20, 2017*. 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.

Frederic Lardinois. 2017. With Cosmos DB, Microsoft wants to build one database to rule them all. https://techcrunch.com/2017/05/10/with-cosmos-db-microsoft-wants-to-build-one-database-to-rule-them-all (Accessed on 23/06/2021).

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

Hongjin Liang and Xinyu Feng. 2021. Abstraction for conflict-free replicated data types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 636–650. https://doi.org/10.1145/3453483.3454067

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)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, 217–237. https://doi.org/10.1007/978-3-319-63390-9_12

Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Peter Müller (Ed.), Vol. 12075. Springer, 544–571. https://doi.org/10.1007/978-3-030-44914-8_20

Chris Newcombe. 2014. Why Amazon Chose TLA +. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*. 25–39. https://doi.org/10.1007/978-3-662-43652-3_3

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015), 66–73. https://doi.org/10.1145/2699417

Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

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)*, Jan Vitek (Ed.), Vol. 9032. Springer, 158–182. https://doi.org/10.1007/978-3-662-46669-8_7

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL (2018), 28:1–28:30. https://doi.org/10.1145/3158116

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)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Amin Timany and Lars Birkedal. 2021. Reasoning about monotonicity in separation logic. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. 91–104. https://doi.org/10.1145/3437992.3439931

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

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue (proof pearl). In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. 76–90. https://doi.org/10.1145/3437992.3439930

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

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 357–368. https://doi.org/10.1145/2737924.2737958

Niklaus Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM* 14, 4 (1971), 221–227. https://doi.org/10.1145/362575.362577

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, Jeremy Avigad and Adam Chlipala (Eds.). ACM, 154–165. https://doi.org/10.1145/2854065.2854081

Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings (Lecture Notes in Computer Science)*, Erika Ábrahám and Catuscia Palamidessi (Eds.), Vol. 8461. Springer, 33–48. https://doi.org/10.1007/978-3-662-43613-4_3

A OPERATIONAL SEMANTICS OF ANERISLANG

In AnerisLang, the state σ is of the form $(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{P}, \mathcal{M})$ where $\mathcal{M}$ is a message soup, a multiset of messages in transit, and $\vec{\mathcal{H}}$, $\vec{\mathcal{S}}$, and $\mathcal{P}$ are mappings from IP addresses to, respectively, node-local heaps $\mathcal{H}$, node-local socket mappings $\mathcal{S}$, and node-local ports in use $\mathcal{P}$. For the relation $(e; \sigma) \rightarrow (e'; \sigma')$ each step either occurs on the network (a system step) or at some particular node (uniquely determined by its IP address). We define steps that take place on nodes by defining a so-called *head-steps* and closing them under evaluation contexts:

$$\frac{(e, \sigma) \rightarrow^h (e', \sigma', (e_{f_1}, \cdots, e_{f_k}))}{(K[e]; \sigma) \rightarrow (K[e'], \sigma', (e_{f_1}, \cdots, e_{f_k}))}$$

We split the head step relation into two parts: head steps that do not modify the network-related state (sockets, ports, and the message soup), and the head steps that do modify the network-related state. We call the latter steps for *network-aware* head steps, and in this section we focus on these; the steps that do not modify the network state are mostly standard. Figure 8 shows reduction rules for the network-aware head step $\rightarrow_{ip}$ (ip being the IP address of the node executing the head step) – note that none of these steps forks any threads and hence we have omitted forked threads in the form of network-aware head steps. Note how the network-aware head step relation does not have access to the heaps part ($\vec{\mathcal{H}}$) of the state.

A socket mapping $\mathcal{S}$ associates each socket handler (file descriptor) z to a UDP-socket represented by a pair $((ip, p), b)$ and a receiving message buffer $\mathcal{B}$. In the pair $((ip, p), b)$, the address (ip, p) is the one to which the socket handler z is bound, and the Boolean b indicates whether the socket is in a blocking (`true`) or non-blocking (`false`) receive mode. Initially, all sockets are in blocking receive mode (*cf.*, Socket-Bind). The rules Set-Socket-To-Non-Blocking-Receive and Set-Socket-To-Blocking-Receive change the receive mode from blocking to non-blocking using a timeout (n, m) (which represents a float $n.m$).[8] The receive buffer $\mathcal{B}$ stores the set of

[8]Timeouts with concrete numbers are strictly speaking not needed as we do not model time in AnerisLang; we use timeouts to align `setReceiveTimeout` with the OCaml function Unix.setsockopt_float.

messages delivered to the node at socket address (ip, p). When a message m is in buffer $\mathcal{B}$ the Receive-Some-Message rule applies and the message m can get delivered to the user application while getting removed from the buffer. When the buffer is empty, one of two network-aware rules applies, depending on whether the socket of the buffer is in the non-blocking (Receive-Empty-Buffer-Non-Blocking) or blocking (Receive-Empty-Buffer-Blocking) receive mode. In the former case, the reduction does not block and it may return `None`. In the latter case, the reduction blocks in the sense that the call to the receive function reduces to itself. Note that his makes the blocking `receivefrom` operation non-atomic. We call such operations stuttering-atomic operations: they reduce to themselves (stutter) a number of times before taking an atomic step. We have shown that the rules applying to atomic steps, *i.e.*, accessing invariants and taking steps in the model, also apply to stuttering atomic steps.

New-Socket
$$\frac{z \notin \text{dom}(\mathcal{S}) \qquad \mathcal{S}' = \mathcal{S}[z \mapsto (\text{None}, \emptyset)]}{(\texttt{socket}\ (), (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (z, (\mathcal{S}', \mathcal{P}, \mathcal{M}))}$$

Socket-Bind
$$\frac{\mathcal{S}(z) = (\text{None}, \emptyset) \qquad p \notin \mathcal{P}(ip) \qquad \mathcal{S}' = \mathcal{S}[z \mapsto (\text{Some}((ip, p), \texttt{true}), \emptyset)] \qquad \mathcal{P}' = \mathcal{P}[ip \mapsto \mathcal{P}(ip) \cup \{p\}]}{(\texttt{socketbind}\ z\ (ip, p), (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (0, (\mathcal{S}', \mathcal{P}', \mathcal{M}))}$$

Send-Message
$$\frac{\mathcal{S}(z) = (\text{Some}((ip, p), \texttt{true}), \mathcal{B}) \qquad \mathcal{M}' = \mathcal{M} \uplus \{((ip, p), to, msg)\}}{(\texttt{sendto}\ z\ msg\ to, (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (|msg|, (\mathcal{S}, \mathcal{P}, \mathcal{M}'))}$$

Receive-Some-Message
$$\frac{to = (ip, p) \qquad m = (from, to, msg) \qquad \{m\} \in \mathcal{B} \qquad \mathcal{S}(z) = (\text{Some}(to, b), \mathcal{B}) \qquad \mathcal{S}' = \mathcal{S}[z \mapsto (\text{Some}(to, b), \mathcal{B} - \{m\})]}{(\texttt{receivefrom}\ z, (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (\texttt{Some}\ (msg, from), (\mathcal{S}', \mathcal{P}, \mathcal{M}))}$$

Receive-Empty-Buffer-Non-Blocking
$$\frac{\mathcal{S}(z) = (\text{Some}((ip, p), \texttt{false}), \emptyset)}{(\texttt{receivefrom}\ z, (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (\texttt{None}, (\mathcal{S}, \mathcal{P}, \mathcal{M}))}$$

Receive-Empty-Buffer-Blocking
$$\frac{\mathcal{S}(z) = (\text{Some}((ip, p), \texttt{true}), \emptyset)}{(\texttt{receivefrom}\ z, (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (\texttt{receivefrom}\ z, (\mathcal{S}, \mathcal{P}, \mathcal{M}))}$$

Set-Socket-To-Non-Blocking-Receive
$$\frac{\mathcal{S}(z) = (\text{Some}((ip, p), b), \mathcal{B}) \qquad (0 \le m \wedge 0 \le n \wedge 0 < (m + n)) \qquad \mathcal{S}'(z) = \mathcal{S}[z \mapsto (\text{Some}((ip, p), \texttt{false}), \mathcal{B})]}{(\texttt{setReceiveTimeout}(m, n), (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} ((), (\mathcal{S}', \mathcal{P}, \mathcal{M}))}$$

Set-Socket-To-Blocking-Receive
$$\frac{\mathcal{S}(z) = (\text{Some}((ip, p), b), \mathcal{B}) \qquad \mathcal{S}'(z) = \mathcal{S}[z \mapsto (\text{Some}((ip, p), \texttt{true}), \mathcal{B})]}{(\texttt{setReceiveTimeout}(0, 0), (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} ((), (\mathcal{S}', \mathcal{P}, \mathcal{M}))}$$

Fig. 8. The rules for network-aware head reduction.

Another way the machine can take a step in AnerisLang is through the system-step relation $\rightarrow_{\mathsf{sys}} \subseteq State \times State$. As explained in the Section 2 of the paper, system steps are those that do not correspond to actual program execution steps. The system step relation in AnerisLang has just two

steps as seen in Figure 9. The Message-Deliver rule corresponds to the system step that delivers a message $(from, to, msg)$ sent from socket address $from$ to the socket address $to = (ip, p)$. Messages are delivered when they are moved into the corresponding receive buffer. The Message-Drop rule corresponds to the network dropping a message.

Message-Deliver
$$\frac{\begin{array}{c} to = (ip, p) \qquad m = (from, to, msg) \in \mathcal{M} \\ \vec{\mathcal{S}}(ip) = \mathsf{Some}(\mathcal{S}) \qquad \mathcal{S}(z) = (\mathsf{Some}(to, b), \mathcal{B}) \\ \vec{\mathcal{S}}' = \vec{\mathcal{S}}[ip \mapsto \mathcal{S}[z \mapsto (\mathsf{Some}(to, b), \mathcal{B} \cup \{m\})]] \end{array}}{(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{P}, \mathcal{M}) \rightarrow_{\mathsf{sys}} (\vec{\mathcal{H}}, \vec{\mathcal{S}}', \mathcal{P}, \mathcal{M})}$$

Message-Drop
$$\frac{m \in \mathcal{M}}{(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{P}, \mathcal{M}) \rightarrow_{\mathsf{sys}} (\vec{\mathcal{H}}, \vec{\mathcal{S}}', \mathcal{P}, \mathcal{M} \setminus \{m\})}$$

Fig. 9. The rules for configuration head reduction.

Note that the operational semantics of AnerisLang described here is different from the original operational semantics of AnerisLang presented in Krogh-Jespersen et al. [2020]. The difference is that the new operational semantics, as explained above, now features blocking receive operations as well as separate system steps for dropping and delivering messages. The old operational semantics did not remove dropped messages from the message soup; it simply ignored them. Also, the old operational semantics did not feature buffers for sockets; any message in the message soup could simply be delivered upon performing a receive operation. The motivation for these changes is to make the operational semantics more realistic.

B TWO-PHASE COMMIT

The two-phase commit protocol [Gray 1978] is one of the best-known practical algorithms for solving the *transaction-commit problem* where a collection of processes, called *resource managers*, have to agree on whether a transaction ought to be *committed* or *aborted*. A protocol that solves the problem has to ensure *agreement*, *i.e.*, no two processes may decide differently.

In this development, we consider a TLA+ model of transaction commit and show that a distributed implementation of the two-phase commit protocol refines it. As a corollary of the refinement and the agreement theorem for the model we show that the implementation also ensures agreement.

Model. The transaction commit model is summarized in Figure 10. The model is parameterized by a set *RMs* of resource managers that are each in either an initial Working state, a preparation state Prepared, or in a final state Committed or Aborted. The full state of the model is a finite mapping from resource managers to one of these states. The transition relation allows resource managers to transition freely from the Working to the Prepared state (TC-Prepare). A resource manager may transition from the Prepared to the Committed state if all resource managers are either in the Prepared or the Committed state (TC-Commit), and from the Working or Prepared state to the Aborted state if no resource manager is in the Committed state (TC-Abort).

By induction on the transition relation one can easily show that the model satisfies the agreement property when starting from an initial state where all resource managers are in the Working state.

Theorem B.1 (Agreement of TC). *Let* $\delta_{\mathsf{Init}}(r) \triangleq \mathsf{Working}$. *If* $\delta_{\mathsf{Init}} \rightharpoonup^*_{\mathsf{TC}} \delta$ *then for all* $r_1, r_2 \in$ *RMs it is* not *the case that* $\delta(r_1) = \mathsf{Committed}$ *and* $\delta(r_2) = \mathsf{Aborted}$.

$$\mathrm{RMStates} \triangleq \{\mathsf{Working}, \mathsf{Prepared}, \mathsf{Committed}, \mathsf{Aborted}\}$$

$$\delta \in RMs \xrightarrow{\mathrm{fin}} \mathrm{RMStates}$$

$$\mathrm{CanCommit}(\delta) \triangleq \forall r \in RMs.\, \delta(r) = \mathsf{Prepared} \lor \delta(r) = \mathsf{Committed}$$

$$\mathrm{NotCommitted}(\delta) \triangleq \forall r \in RMs.\, \delta(r) \neq \mathsf{Committed}$$

TC-Prepare
$$\frac{\delta(r) = \mathsf{Working}}{\delta \rightharpoonup_{\mathsf{TC}} \delta[r \mapsto \mathsf{Prepared}]}$$

TC-Commit
$$\frac{\delta(r) = \mathsf{Prepared} \qquad \mathrm{CanCommit}(\delta)}{\delta \rightharpoonup_{\mathsf{TC}} \delta[r \mapsto \mathsf{Committed}]}$$

TC-Abort
$$\frac{\delta(r) = \mathsf{Working} \lor \delta(r) = \mathsf{Prepared} \qquad \mathrm{NotCommitted}(\delta)}{\delta \rightharpoonup_{\mathsf{TC}} \delta[r \mapsto \mathsf{Aborted}]}$$

Fig. 10. TLA$^+$ specification of the transaction-commit (TC) problem.

Implementation. The two-phase commit protocol relies on a *transaction manager* to orchestrate the agreement process; the transaction manager may either be a distinguished resource manager or a separate process. Listing 4 and Listing 5 show implementations in AnerisLang of the transaction manager and resource manager roles, respectively. The transaction manager uses a simple library functionality for removing duplicate messages; the functionality is initialized with the `nodup_init` invocation that returns a wrapped `receivefrom` primitive that removes duplicates messages but is otherwise not important.

Listing 4. Transaction manager.

```
let recv_resps recv skt RMs =
  let rec loop prepared =
    Set.equal prepared RMs ||
      let (msg, sndr) = recv skt in
      msg = "PREPARED" &&
        loop (Set.add sndr prepared) in
  loop (Set.empty ()) in

let transaction_manager TM RMs =
  let skt = socket () in
  socketbind skt TM;
  let recv = nodup_init () in
  sendto_all skt RMs "PREPARE";
  let ready = recv_resps recv skt RMs in
  if ready then
    sendto_all skt RMs "COMMIT";
    receivefrom_all skt recv RMs;
    "COMMITTED"
  else
    sendto_all skt RMs "ABORT";
    "ABORTED"
```

Listing 5. Resource manager.

```
let resource_manager RM TM =
  let skt = socket () in
  socketbind skt RM;
  let (m, _) = receivefrom skt in
  if m = "ABORT"
  then sendto skt "ABORTED" TM
  else
    let local_abort = coin_flip () in
    if local_abort
    then sendto skt "ABORTED" TM
    else
      sendto skt "PREPARED" TM;
      let (decision, _) =
        wait_receivefrom skt
        (fun (_, m) => m = "COMMIT" ||
                       m = "ABORT") in
      if decision = "COMMIT" then
        sendto skt "COMMITTED" TM
      else
        sendto skt "ABORTED" TM
```

The transaction manager implementation starts by allocating a socket and binding it to the socket address `TM` given as argument. It continues by sending a `"PREPARE"` message to all the resource manager socket addresses given in `RMs`, asking the resource managers to transition to the preparation phase. If all the resource managers respond with `"PREPARED"`—signifying that they are all ready to commit—the transaction manager continues by sending a `"COMMIT"` message, telling the resource managers the decision is to commit, after which it awaits their responses and returns.

If a single resource manager responds with `"ABORTED"`, the transaction manager stops receiving responses, relays the information, and returns.

The resource manager implementation starts by allocating a socket and binding it to the socket address `RM` given as argument. It continues by listening for an initial request from the transaction manager; in case another resource manager already aborted and this information arrived prior to the initial `"PREPARE"` request, the resource manager aborts. If asked to prepare, the resource manager makes a local decision—here with a nondeterministic coin flip–and sends the decision to the transaction manager. If the resource manager decides to abort, it immediately returns; otherwise it awaits the final decision from the transaction manager, confirms the transition, and returns.

Refinement. To show that the two-phase commit implementation refines the transaction-commit model we instantiate the Aneris logic with the model; this gives us a handle to the current model state δ that we can manipulate through the separation logic resource $\mathsf{Model}_{\circ}(\delta)$. The key proof strategy is to keep this resource in an invariant that ties together the model state and the physical with enough information such that the continued simulation is strong enough for proving our final correctness theorem (Theorem B.2). In this development, we will tie sending a message (such as `"COMMITTED"`) from resource manager r to the corresponding transition in the model (such as TC-Commit). Additionally, the invariant will have to keep sufficient ghost resources and information for us to establish the conditions ($\mathrm{CanCommit}(\delta)$ and $\mathrm{NotCommitted}(\delta)$) for progressing the model.

To state a sufficient invariant for the two-phase commit refinement we will make use of two resource algebras: a variation of the *oneshot* algebra [Jung et al. 2018] with *discardable fractions* [Vindum and Birkedal 2021] as well as a monotone ghost map algebra.

The oneshot algebra with discardable fractions allows us to define resources $pending(q)$, *discarded*, and $shot(a)$ governed by the rules below.

$$
\begin{array}{ll}
pending(q) \Rrightarrow discarded & pending(1) \Rrightarrow shot(a) \\
shot(a) * pending(q) \vdash \mathsf{False} & pending(p) * pending(q) \dashv\vdash pending(p+q) \\
shot(a) * discarded \vdash \mathsf{False} & shot(a) * shot(a) \dashv\vdash shot(a) \\
shot(a) * shot(b) \vdash a = b & discarded * discarded \dashv\vdash discarded
\end{array}
$$

Intuitively, $pending(q)$ corresponds to owning a q-sized share in making some decision; only by owning all shares a unique decision can be made as witnessed by owning $shot(a)$. By discarding a share a party can ensure that no decision can ever be made. Notice how this construction can be used to model the two-phase commit protocol (in particular the condition $\mathrm{CanCommit}(\delta)$ and $\mathrm{NotCommitted}(\delta)$) by picking the decision value a to be the unit value: each party initially owns an evenly sized share of the decision and transfers this share to the transaction manager when preparing to commit. By receiving a share from all resource managers, the transaction manager can make the decision to commit. By discarding a share, a resource manager can ensure that no decision to commit will ever be made and safely abort.

Using the *monotone resource algebra* [Timany and Birkedal 2021], we construct a logical points-to connective $r \overset{q}{\mapsto}_{\bullet} s$ that will track q-fractional ownership of the current model state s of resource manager r, but where s may only evolve monotonically according to the internal resource manager transition relation given by $\mathsf{Working} \rightsquigarrow \mathsf{Prepared} \rightsquigarrow \mathsf{Committed}$ and $\mathsf{Working}, \mathsf{Prepared} \rightsquigarrow$

Aborted. The construction is accompanied by a duplicable $r \mapsto_\circ s$ resource that gives a *lower-bound* on the current state of resource manager r as seen from the rules below.

$$r \overset{q}{\mapsto}_\bullet s * r \mapsto_\circ s' \vdash s' \rightsquigarrow^* s$$

$$r \overset{1}{\mapsto}_\bullet s * s \rightsquigarrow^* s' \Rrightarrow r \overset{1}{\mapsto}_\bullet s' * r \mapsto_\circ s'$$

$$r \mapsto_\circ s * r \mapsto_\circ s \dashv\vdash r \mapsto_\circ s$$

Equipped with the two constructions from above we can define the refinement invariant for the two-phase commit implementation:

$$I_{\mathsf{TPC}} \triangleq \exists \delta.\, \mathsf{Model}_\circ(\delta) * \mathop{\bigast}_{r \in RMs} \exists R, T, s.\ \begin{array}{l} r \overset{\frac{1}{2}}{\mapsto}_\bullet s * \delta(r) = s * \mathrm{TokenCoh}(s) * \\ r \rightsquigarrow_\square^{\phi_{\mathsf{RM}}} (R, T) * \mathrm{ModelCoh}(r, s, T) \end{array}$$

The invariant owns the current model state δ and for each resource manager r it owns half of the corresponding monotone points-to connective for some state s such that $\delta(r) = s$; the resource manager itself will own the remaining half. This ensures that the resource manager itself knows exactly which state it is in and that the resource cannot be updated without updating the model as well. We moreover tie being in the model states Committed and Aborted to ownership of, respectively, the *shot* and *discarded* resources as given by $\mathrm{TokenCoh}(s)$ below.

$$\mathrm{TokenCoh}(s) \triangleq \begin{cases} \mathit{shot} & \text{if } s = \mathsf{Committed} \\ \mathit{discarded} & \text{if } s = \mathsf{Aborted} \\ \mathsf{True} & \text{otherwise} \end{cases}$$

The remaining two clauses constitute the key component in connecting the model to the physical state; the persistent socket protocol $r \rightsquigarrow_\square^{\phi_{\mathsf{RM}}} (R, T)$ tracks the history T of sent messages from resource manager r and $\mathrm{ModelCoh}(r, s, T)$ requires that if the resource manager r is in state s then a corresponding message must have been sent to the transaction manager t and if a message corresponding to a state s' has been sent, the resource manager must be in *at least* that state:

$$\mathrm{MessageCoh}(r, s, T) \triangleq \begin{cases} (r, t, \text{"PREPARED"}) \in T & \text{if } s = \mathsf{Prepared} \\ (r, t, \text{"COMMITTED"}) \in T & \text{if } s = \mathsf{Committed} \\ (r, t, \text{"ABORTED"}) \in T & \text{if } s = \mathsf{Aborted} \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathrm{ModelCoh}(r, s, T) \triangleq \mathrm{MessageCoh}(r, s, T) \wedge \forall s'.\, \mathrm{MessageCoh}(r, s', T) \to s' \rightsquigarrow^* s$$

The socket protocol ϕ_{TM} governing the communication with the transaction manager is defined below. It follows the intuitive description given earlier: when preparing to commit, the *pending* resource is transferred to the transaction manager, and in order to commit or abort, the resources *shot* and *discarded* must be transferred as well, respectively. Moreover, the resource manager has to prove that its model state has (at least) been progressed to the corresponding states. The socket protocol for ϕ_{RM} for the resource manager follows a similar pattern.

$$\begin{aligned} \phi_{\mathsf{TM}}(r, t, b) \triangleq\ & r \in RMs * \\ & \left(b = \text{"PREPARED"} * \mathit{pending}(\tfrac{1}{|RMs|+1}) * r \mapsto_\circ \mathsf{Prepared}\right) \vee \\ & (b = \text{"COMMITTED"} * \mathit{shot} * r \mapsto_\circ \mathsf{Committed}) \vee \\ & (b = \text{"ABORTED"} * \mathit{discarded} * r \mapsto_\circ \mathsf{Aborted}) \end{aligned}$$

$$\phi_{\mathsf{RM}}(r, t, b) \triangleq b = \text{"PREPARE"} \vee$$

$$(b = \text{"COMMIT"} * \mathit{shot} * \mathop{\Huge *}_{r \in RMs} r \mapsto_{\circ} \mathsf{Prepared}) \lor$$
$$(b = \text{"ABORT"} * \mathit{discarded})$$

The transaction manager implementation can be given the specification below; notice how it does not rely on the refinement invariant but only on the socket protocols and resources as described.

$$\left\{\begin{array}{l}\mathsf{Fixed}(A) * t \in A * \mathsf{FreePort}(t) * t \rightsquigarrow (\emptyset, \emptyset) * \\ \mathit{pending}(\frac{1}{|RMs|+1}) * t \Mapsto \phi_{\mathsf{TM}} * \mathop{\Huge *}_{r \in RMs} r \Mapsto \phi_{\mathsf{RM}}\end{array}\right\}$$
$$\langle t;\ \mathtt{transaction_manager}\ t\ RMs \rangle$$
$$\left\{ v.\ \begin{array}{l}(v = \text{"COMMITTED"} * \mathop{\Huge *}_{r \in RMs} r \mapsto_{\circ} \mathsf{Committed}) \lor \\ (v = \text{"ABORTED"} * \exists r \in RMs.\, r \mapsto_{\circ} \mathsf{Aborted})\end{array}\right\}$$

The specification for the resource manager as seen below, however, relies on the invariant as well as fractional ownership of the resource manager's model state.

$$\left\{\begin{array}{l}\mathsf{Fixed}(A) * r \in A * \mathsf{FreePort}(r) * \boxed{I_{\mathsf{TPC}}}^{\mathcal{N}_{\mathsf{TPC}}} * \\ r \Mapsto \phi_{\mathsf{RM}} * t \Mapsto \phi_{\mathsf{TM}} * \mathit{pending}(\frac{1}{|RMs|+1}) * r \overset{\frac{1}{2}}{\mapsto}_{\bullet} \mathsf{Working}\end{array}\right\}$$
$$\langle r;\ \mathtt{resource_manager}\ r\ t \rangle$$
$$\{\mathsf{True}\}$$

Theorem B.2 (Agreement, two-phase commit implementation). *If* $(e; \emptyset) \to^* (T; \sigma)$ *and* $m_{s_1}, m_{s_2} \in \mathcal{M}$ *such that* m_{s_i} *is the physical message corresponding to state* s_i *then it is not the case that* $s_1 = \mathsf{Committed}$ *and* $s_2 = \mathsf{Aborted}$.

C SINGLE-DECREE PAXOS

C.1 Auxiliary implementation components

```
let recv_promises skt n bal0 =
  let promises = ref (Set.empty ()) in
  let senders = ref (Set.empty ()) in
  let rec loop () =
    if Set.cardinal !senders = n
    then !promises
    else
      let (m, sndr) = receivefrom skt in
      let (bal, mval) =
        proposer_deser m in
      if bal = bal0 then
        senders <- Set.add !senders sndr;
        promises <- Set.add !promises mval
      else ();
      loop ()
  in loop ()
```

```
let find_max_promise s =
  let max_promise acc promise =
    match promise, acc with
    | Some (b1, _), Some (b2, _) =>
      if b1 < b2 then acc else promise
    | None, Some _ => acc
    | _, _ => promise
    end
  in Set.fold max_promise s None
```

```
let learner acceptors addr =
  let skt = socket () in
  socketbind skt addr;
  let majority =
    Set.cardinal acceptors / 2 + 1 in
  let votes = ref (Map.empty ()) in
  let rec loop () =
    let (m, sndr) = receivefrom skt in
    let (bal, v) = learner_deser m in
    let bal_votes =
      match Map.find_opt bal !votes with
      | Some vs => vs
      | None => Set.empty ()
      end in
      let bal_votes' =
        Set.add sndr bal_votes in
    if Set.cardinal bal_votes' = majority
    then (bal, v)
    else
      votes <- Map.add bal bal_votes' votes;
      loop ()
  in loop ()
```

```
let wait_receivefrom skt test =
  let rec loop () =
    let msg = receivefrom skt in
    if test msg then msg else loop ()
  in loop ()

let sendto_all skt X msg =
  Set.iter (fun x => sendto skt msg x) X
```

C.2 Lifted Paxos TLA model

SDP-INC

$$(C, \delta) \rightharpoonup_{\widehat{\mathsf{SDP}}} (C[p \mapsto C(p) + 1], \delta)$$

SDP-A

$$\frac{C(p) = n \qquad b = n \cdot |\mathsf{Proposer}| + p \qquad \begin{array}{c}(\mathcal{S}, \mathcal{B}, \mathcal{V}) \rightharpoonup_{\mathsf{SDP}} (\mathcal{S} \cup \{m\}, \mathcal{B}, \mathcal{V}) \\ m = \mathsf{msg1a}(b) \lor m = \mathsf{msg2a}(b, v)\end{array}}{(C, \mathcal{S}, \mathcal{B}, \mathcal{V}) \rightharpoonup_{\widehat{\mathsf{SDP}}} (C, \mathcal{S} \cup \{m\}, \mathcal{B}, \mathcal{V})}$$

SDP-B

$$\frac{(\mathcal{S}, \mathcal{B}, \mathcal{V}) \rightharpoonup_{\mathsf{SDP}} (\mathcal{S} \cup \{m\}, \mathcal{B}', \mathcal{V}') \qquad m = \mathsf{msg1b}(a, b, o) \lor m = \mathsf{msg2b}(a, b, v)}{(C, \mathcal{S}, \mathcal{B}, \mathcal{V}) \rightharpoonup_{\widehat{\mathsf{SDP}}} (C, \mathcal{S} \cup \{m\}, \mathcal{B}', \mathcal{V}')}$$

D FAIR TERMINATION OF CONCURRENT PROGRAMS, DETAILS

The section gives a few definitions which were alluded to in Section 5.

D.1 Fairness model

A fairness model is a state transition system, with a set of roles which label its transitions. Each state has a set of enabled roles, and a "fuel limit" which is used to keep the control the branching of $\mathsf{Live}(\mathcal{F})$ which we define below.

Definition D.1. A *fairness model* $\mathcal{F}$ is the data of a set $\mathcal{F}$ of states, a set $\mathcal{R}$ of roles, and a transition relation $\rightarrow \subseteq \mathcal{F} \times \mathcal{R} \times \mathcal{F}$ labeled by roles. Moreover, it is equipped with a map *enabled_roles* : $\mathcal{F} \rightarrow \wp_{\mathrm{fin}}(\mathcal{R})$ which associates a finite set of *enabled roles* to each state $s \in \mathcal{F}$. It must approximate the set of outgoing roles:

$$\forall \rho \in \mathcal{R},\ \forall \delta, \delta' \in \mathcal{F},\ \delta \xrightarrow{\rho} \delta' \implies \rho \in \mathit{enabled_roles}\ \delta.$$

and it must not disable other roles: if $\delta \xrightarrow{\rho} \delta'$, then

$$\forall \rho' \neq \rho, \quad \rho' \in \mathit{enabled_roles}\ \delta \ \Rightarrow\ \rho' \in \mathit{enabled_roles}\ \delta'$$

Finally, $\mathcal{F}$ comprises a map $fuel_limit : \mathcal{F} \to \mathbb{N}$ which will be useful to ensure finite branching conditions.

A *run*, or *trace* of $\mathcal{F}$ is a non-empty finite or infinite sequence of the form:

$$\delta_1 \xrightarrow{\rho_1} \delta_2 \xrightarrow{\rho_2} \cdots$$

D.2 The Live construction

Given a "fairness model" $\mathcal{F}$, we define a (labeled) STS $\mathsf{Live}(\mathcal{F})$ which keeps track of mapping between roles and threads, and of fuels, as explained in Section 5.

A state of $\mathsf{Live}(\mathcal{F})$ is a triple (δ, F, T) of a state $\delta \in \mathcal{F}$, together with two maps $F : enabled_roles\ \delta \to \mathbb{N}$ and $T : enabled_roles\ \delta \to \mathbb{N}$ which associate a fuel amount and a thread id to each role ρ which is enabled in the current underlying state δ.

The set of labels of $\mathsf{Live}(\mathcal{F})$ is

$$\{\mathsf{Step}\ \rho\ \mathsf{tid} \mid \rho \in \mathcal{R}, \mathsf{tid} \in \mathbb{N}\} \quad \uplus \quad \{\mathsf{Silent}\ \mathsf{tid} \mid \mathsf{tid} \in \mathbb{N}\}$$

(recall that $\mathcal{R}$ is the set of roles of the "fairness model" $\mathcal{F}$) The intuition is that a step labeled by $\mathsf{Step}\ \rho\ \mathsf{tid}$ corresponds to the situation where the thread tid takes a step in the program, and one of the roles ρ under its responsibility takes a step in the fairness model. A step labeled $\mathsf{Silent}\ \mathsf{tid}$, on the other hand, corresponds to a step in the program which does not correspond to a step in the fairness model, in other words, a stuttering step.

We now describe the transitions in the labeled STS $\mathsf{Live}(\mathcal{F})$. The idea is that there are two kinds of transitions

- A thread tid can take a step in $\mathsf{Live}(M)$ of the form

  $$(\delta, F, T) \xrightarrow{\mathsf{Silent}\ \mathsf{tid}} (\delta, F', T')$$

  which does not corresponds to a step in the underlying fairness model $\mathcal{F}$ in exchange of consuming fuel: for *every* $\rho \in T^{-1}(\mathsf{tid})$ which the thread tid is in charge of, the corresponding fuel $F(\rho)$ must decrease strictly, in that $F'(\rho) < F(\rho)$.
- A thread tid can also take a step in the underlying model which corresponds to a role $\rho \in T^{-1}(\mathsf{tid})$

  $$(\delta, F, T) \xrightarrow{\mathsf{Step}\ \rho\ \mathsf{tid}} (\delta', F', T')$$

  This allows to refill the fuel of ρ up to the limit $fuel_limit\ \delta'$, that is, $F'(\rho) \leq fuel_limit\ \delta'$; this is required to keep the STS finitely branching. All the other roles which are associated with tid must decrease:

  $$\forall \rho' \in T(\mathsf{tid}) \setminus \{\rho\}, \quad F'(\rho') < F(\rho')$$

  Roles which appear between δ and δ' can have any fuel $\leq fuel_limit\ \delta'$ in F'. Of course, we also require there be a step

  $$\delta \xrightarrow{\rho} \delta'$$

  in the fairness model $\mathcal{F}$.

In addition to the two constraints above, in both cases, the fuel of the roles which are not associated with the thread tid must not increase:

$$\forall \rho \in \mathcal{R} \setminus T^{-1}(\mathsf{tid}), \quad F'(\rho) \leq F(\rho)$$

and roles which change owners ($T'(\rho) \neq T(\rho)$) must have their fuel decrease strictly.

D.3 The $\mathcal{F}_{\text{YN}}$ model

The full $\mathcal{F}_{\text{YN}}$ model is defined as follows. Its states are quadruples

$$(m, b, ye, ne) \in \mathbb{N} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}$$

Its set of roles is {Yes, No}, $fuel_limit$ is the constant map equal to 30. The map $enabled_roles$ is defined so that

$$\mathsf{Yes} \in enabled_roles\ (m, b, ye, ne) \iff ye = 1$$
$$\mathsf{No} \in enabled_roles\ (m, b, ye, ne) \iff ne = 1$$

It remains to define the transitions. First, there are the main types of transition which is described in Figure 7:

1. success for Yes: $(m, 1, 1, 1) \xrightarrow{\mathsf{Yes}} (m, 0, 1, 1)$ if $m > 0$
2. failure for Yes: $(m, 0, 1, 1) \xrightarrow{\mathsf{Yes}} (m, 0, 1, 1)$ if $m > 0$
3. success for No: $(m, 0, 1, 1) \xrightarrow{\mathsf{No}} (m - 1, 1, 1, 1)$ if $m > 0$
4. failure for No: $(m, 1, 1, 1) \xrightarrow{\mathsf{No}} (m, 1, 1, 1)$

and the three transitions to account for "shutting down" the threads:

5. the last transition of No after Yes has finished: $(1, 0, 0, 1) \xrightarrow{\mathsf{Yes}} (0, 1, 0, 1)$;
6. Yes terminates: $(m, b, 1, ne) \xrightarrow{\mathsf{Yes}} (m, b, 0, ne)$ if $m \leq 1$;
7. No terminates: $(0, b, ye, 1) \xrightarrow{\mathsf{No}} (0, b, ye, 0)$ if $m \leq 1$.

It is easy to check that the only transition which do not decrease the state, ordered with lexicographic order on (m, b) and the product order on $((m, b), ye, ne)$ are the two loop transitions. Moreover, in a state (m, b, ye, ne), all the transitions labeled with "if $b = 1$ then Yes else No" decrease the state. This means that the following criterion shows the model is fairly terminating.

D.4 Locally fairly terminating models

Definition D.2. A fairness model $\mathcal{F}$ is called *locally fairly terminating* if there exists a well founded order $\leq$ over $\mathcal{F}$ and a map $\pi : \mathcal{F} \to \mathcal{R}$ from states to roles which satisfies the following conditions:

1. for all transitions $\delta \xrightarrow{\rho} \delta'$, $\delta' \leq \delta$;
2. for all states $\delta \in \mathcal{F}$ which are not dead ends, $\pi(\delta) \in enabled_roles\ \delta$ and for all δ' such that $\delta \xrightarrow{\pi(\delta)} \delta'$, $\delta' < \delta$;
3. for all transitions $\delta \xrightarrow{\rho} \delta'$, if $\rho \neq \pi(\delta)$, then $\pi(\delta') = \pi(\delta)$;

where a state δ is called a dead end if there are no outgoing transitions from it.

We call this criterion *local* because it can be checked for each transition independently, without any reference to traces. This criterion is correct:

Lemma D.3. *If a fairness model $\mathcal{F}$ is locally fairly terminating, then it is fairly terminating.*