Trillium: History-Sensitive Refinement in Separation Logic

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

Formal verification systems such as TLA$^+$ have been widely used to design, model, and verify complex concurrent and distributed systems. In many of these tool suites, systems are modeled as state transition systems, and both safety and liveness properties can usually be checked. This enables users to reason about abstract system specifications and uncover design flaws, but it offers no guarantees about the *implementations* of systems nor about the relation between an implementation and its abstract specification.

In this work, we show how to connect concrete implementations of concurrent and distributed systems to abstract system models. We develop Trillium, a separation logic framework for establishing *history-sensitive* refinement relations between programs and models. We use our logic to prove correctness of 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. This 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

Formal verification systems such as SPIN [Holzmann 1997] and TLA$^+$ [Lamport 1992] have been widely used to design, model, and verify complex concurrent and distributed systems with successful industrial applications in organizations like NASA [Havelund et al. 2001], Intel [Beers 2008], Amazon [Newcombe 2014; Newcombe et al. 2015], and Microsoft [Lardinois 2017]. In many of these tool suites, systems are modeled as state transition systems, and they can usually check both safety and liveness properties. This enables users to reason about abstract system specifications and uncover design flaws, but it offers no guarantees about the *implementations* of systems nor about the relation between an implementation and its abstract specification.

Concurrent separation logic [O'Hearn 2007] and its modern variants, such as Iris [Jung et al. 2016, 2018, 2015; Krebbers et al. 2017a], provide powerful and modular reasoning principles for concurrent programs, thanks to mechanisms such as *ghost state* and *invariants*. They have been used to verify a wide range of implementations of sophisticated fine-grained concurrent data structures [Carbonneaux et al. 2022; Chajed et al. 2021; de Vilhena et al. 2020; Krishna et al. 2020; Mével and Jourdan 2021; Vindum and Birkedal 2021; Vindum et al. 2022] and distributed systems [Gondelman et al. 2021; Krogh-Jespersen et al. 2020].

In this work, we show how to connect concrete implementations of concurrent and distributed systems to abstract system models. We develop Trillium, a modular language-agnostic separation logic framework for establishing *history-sensitive* refinement relations between programs and models, which relate *traces* of program executions to *traces* of a model. Not only does the refinement

Authors' addresses: Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Simon Oddershede Gregersen, Aarhus University, Denmark, gregersen@cs.au.dk; Léo Stefanesco, MPI-SWS, Germany, leo.stefanesco@gmail.com; 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.

```
let rec inc_loop =
  let n = !ℓ in
  cas(ℓ, n, n + 1);
  inc_loop ()
in
  inc_loop () || inc_loop ()
```

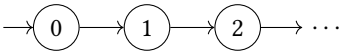


Fig. 1. A concurrent counter inc and its corresponding model $\mathcal{M}_{\mathsf{inc}}$.

relation establish a formal connection and hence a specification of the program in its own right, but it makes it possible to transfer both safety and liveness properties of a model to its implementation. Consequently, this allows us to exploit existing properties of abstract models to establish properties about programs that are not expressible using ordinary concurrent separation logics.

Our development is *foundational* [Appel 2001], in that all our results, including the operational semantics, the models, and the logic, are formalized in the Coq proof assistant on top of the Iris base logic while using the Iris Proof Mode [Krebbers et al. 2017b] for reasoning within the logic. In §3 we show how concrete implementations of two distributed protocols, *two-phase commit* [Gray 1978] and *single-decree Paxos* [Lamport 1998, 2001], formally refine their abstract TLA$^+$ specifications, and how safety properties of the models can be transferred to the implementations. Moreover, we explain in §4 and §5 how to prove liveness properties of distributed and concurrent programs through refinement: we prove strong eventual consistency of an implementation of a *Conflict-Free Replicated Data Type* (CRDT) [Shapiro et al. 2011] and fair termination of a concurrent program.

The Trillium Methodology. Consider the example in Figure 1 that shows a program inc written in an ML-like imperative language. The program uses a global reference ℓ, with an initial value of zero, that it increments concurrently in two infinite loops using *compare-and-set* operations. Our goal is to prove that the reference ℓ takes successively the values 0, 1, 2, ... without skipping any number. To do so, we will establish a refinement between the program and an abstract model $\mathcal{M}_{\mathsf{inc}}$, depicted on the right-hand side of Figure 1. In general, a *model* in Trillium is an arbitrary state transition system $\mathcal{M} = (A_{\mathcal{M}}, \rightharpoonup_{\mathcal{M}})$ where $A_{\mathcal{M}}$ denotes a set of states and $\rightharpoonup_{\mathcal{M}} \subseteq A_{\mathcal{M}} \times A_{\mathcal{M}}$ the transition relation on states. In this particular model, at each step, either the state does not change, or it is increased by one. The refinement will express that, at all times, the value stored at location ℓ is equal to the current value of the model and no value is skipped. Note that prior techniques for proving refinements in Iris could not express the fact that the program never skips a number (see §6.1 for a more elaborate discussion) and that our techniques also allow us to prove the same property about programs that allocate locations dynamically (see the supplementary material).

As Trillium is a *separation logic*, propositions denote not only facts but *ownership* of resources. For example, the proposition $\ell \mapsto v$ asserts exclusive ownership of location ℓ storing value v. When, as in the example of Figure 1, two threads access the same location, the resource P can be shared by placing it into an *invariant*, *e.g.*, $\boxed{P}$. Invariants are guaranteed to hold a every computation step and they are duplicable, *i.e.*, $\boxed{P} \vdash \boxed{P} * \boxed{P}$, which means invariants can be shared among multiple threads. The *separating conjunction* $P * Q$ holds for resources that can be split into two disjoint sub-resources which satisfy P and Q respectively.

When instantiated with a programming language, Trillium provides a new notion of Hoare triple $\{P\}\, e\, \{Q\}^{\mathcal{M}}$ and a special resource $\mathsf{Model}_\circ\,(\delta : \mathcal{M})$ that represents the current state of the abstract model $\mathcal{M}$ and asserts that it is equal to δ. We omit $\mathcal{M}$ from the connectives when it is clear from the context. The refinement we want to show can be established in Trillium by proving, using the

logical predicates and rules of the Trillium logic, the validity of the following Hoare triple:

$$\left\{\boxed{\exists n.\, \ell \mapsto n * \mathsf{Model}_\circ(n)}\right\}\ \mathtt{inc}\ \{\mathsf{False}\}^{\mathcal{M}_{\mathsf{inc}}}$$

where `inc` is the example program. The invariant ties the contents of the reference ℓ to the current state of the model and enforces that they always agree. The postcondition of the Hoare triple is False as the program does not terminate.

Intuitively, a Hoare triple like the above means the program is *safe*, *i.e.*, it does not crash, and that the post condition holds for its end-state. To prove this formally, Trillium comes with an *adequacy theorem*. In Trillium, however, the adequacy theorem additionally concludes a history-sensitive refinement relation between all traces generated by the program and the model and, as such that, that the model state and the reference ℓ progress in a lock-step fashion.

Definition 1.1 (History-Sensitive Refinement). Let τ be a program execution trace, κ a model trace, and ξ a binary relation on traces. τ is a *history-sensitive refinement* of κ under ξ whenever $\tau \precsim_\xi \kappa$ holds, where $\tau \precsim_\xi \kappa$ is coinductively defined by:

$$\tau \precsim_\xi \kappa \triangleq \xi(\tau, \kappa) \land \forall c.\, last(\tau) \rightarrow c \Rightarrow \exists \delta.\, \tau :: c \precsim_\xi \kappa :: \delta$$

where :: denotes trace extension, and $\rightarrow$ is the stepping-relation of the operational semantics of the programming language.

That is, an execution trace τ is a history-sensitive refinement of a model trace κ under ξ if $\xi(\tau, \kappa)$ holds and for all configurations that the last configuration of τ may step to, there exists a model state δ such that the extended traces refine each under ξ as well. This relation straightforwardly extends to all finite prefixes of all possibly-infinite traces generated by program execution as detailed in §6.1. This will be crucial for properties, such as liveness, that we consider. Figure 2 illustrates graphically history-sensitive refinement.

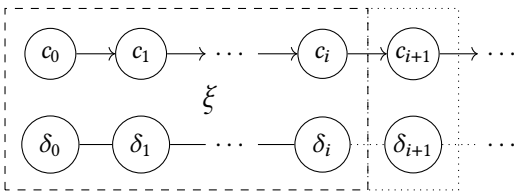


Fig. 2. Illustration of Definition 1.1. Trace $\tau = c_0 :: c_1 :: \ldots :: c_i$ refines $\kappa = \delta_0 :: \delta_1 :: \ldots :: \delta_i$ under ξ, written $\tau \precsim_\xi \kappa$ if $\xi(\tau, \kappa)$ and for any c_{i+1} that c_i may step to there exists δ_{i+1} such that $\tau :: c_{i+1} \precsim_\xi \kappa :: \delta_{i+1}$ holds.

For our running example, we pick the predicate ξ_{inc} to express that the value of the reference ℓ in the current state of the program (the last state of the trace) is equal to the current state of the model and does not skip model states:

$$\xi_{inc}(\tau, \kappa) \quad \triangleq \quad heap(last(\tau))(\ell) = last(\kappa) \land stuttering(\kappa)$$

where the operator *last* returns the last element of a (non-empty) trace, $heap(c)$ returns the heap component of a program configuration c, and

$$stuttering(\kappa) = \begin{cases} last(\kappa') = \delta \lor last(\kappa') \rightarrow_{\mathcal{M}_{\mathsf{inc}}} \delta & \text{if } \kappa = \kappa' :: \delta \\ \mathsf{True} & \text{otherwise} \end{cases}$$

In this simple example, the relation ξ_{inc} only depends on the last elements of the two traces, in which case history-sensitive refinement reduces to the usual notion of simulation. We will see in §4 and §5 situations where the additional expressive power of history-sensitive relations is needed.

We can now state a simplified version of the adequacy theorem, saving the full statement of the adequacy theorem for §6.

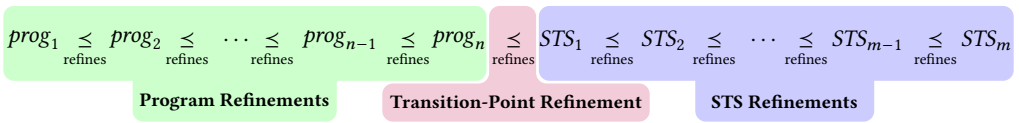
Theorem 1.2 (Simplified adequacy). *Let e be a program, σ_0 a program state, and Φ an Iris predicate on values. Let $\delta_0 \in \mathcal{M}$ be a model state and ξ a finitary binary relation on program and model traces. Suppose*

$$\left\{\boxed{P_1} * \cdots * \boxed{P_k}\right\} e \left\{\Phi\right\}^{\mathcal{M}} \quad \text{and} \quad \mathit{AlwaysHolds}(\xi, e, \sigma_0, \delta_0)$$

can be proved in the Trillium logic, *then $(e, \sigma_0) \precsim_\xi \delta_0$ holds in the meta-logic, e.g. Coq, and the program e is safe when started in the initial heap σ_0. The Trillium predicate $\mathit{AlwaysHolds}(\xi, e, \sigma_0, \delta_0)$ expresses that the invariants $\boxed{P_1}, \ldots, \boxed{P_k}$ imply that ξ holds at the current state.*

We emphasize that the Trillium Hoare triple is an extension of the usual Iris-style Hoare triple and we can seamlessly reuse Iris program logic proofs in Trillium. The proof rules of Trillium include all the usual rules of program logics built on top of Iris; there is just one additional proof rule for reasoning about refinement. It also means that the *same* Hoare triple can be used to capture refinement as well as functional correctness. In particular, it is possible to use properties of the model when proving a Hoare triple as it is embedded in the logic as a resource. The specification can, *e.g.*, be used by clients to show safety, which might rely on properties of the model; see §3.2 for an example and a more detailed discussion.

Composing Refinements. One of the aspects that makes refinements a powerful tool in studying programs and computer systems is their composability. That is, one can prove that a system A refines B and that B refines C in order to establish A refines C. This concept is useful especially in cases where the gap, *e.g.*, in implementation details, between A and B, and between B and C, are smaller than the gap between A and C which makes those refinements easier to establish. What we are interested in, especially in verification of distributed systems, is a tower of composable refinements where one end is a low-level, realistic program implementation while the other end is a high-level, abstract state transition system (STS) that captures the essence of the implementation. That is, we are interested in a tower of refinements of the form:



The literature on reasoning about both programs and systems includes many works on proving refinements between pairs of programs [Birkedal et al. 2012; Krebbers et al. 2017c; Turon et al. 2013a,b], including reasoning in program logics, and pairs of STSs [Lamport 1992]. In this work, we focus on the *transition-point refinement* in the diagram above where we only consider refinements between programs and STSs. See §7 for a comparison of the present work with other works that also involve transition-point refinements. In this regard, Trillium enables incorporation of different tools as different refinements need not be proven in the same formalism in order for them to be composed. An example of this fact is illustrated in our example where we show refinement between the implementation of single-decree Paxos and its TLA$^+$ spec (see §3). The TLA$^+$ spec that we use in this example is in fact the low-level TLA$^+$ spec which itself is shown to refine another high-level TLA$^+$ model.[1] Hence, our proof also establishes that the single-decree Paxos implementation we consider refines this more high-level TLA$^+$ model. We believe, but have not formally established,

[1]See https://github.com/tlaplus/Examples/blob/master/specifications/PaxosHowToWinATuringAward/Paxos.pdf and https://github.com/tlaplus/Examples/blob/master/specifications/PaxosHowToWinATuringAward/Voting.pdf.

that Trillium can be combined with other approaches for refinements between a pair of programs, *e.g.*, Krebbers et al. [2017c], in order to establish a refinement between a program and an STS.

Instantiations. In §6.2 we instantiate Trillium with a subset of OCaml, AnerisLang, a distributed higher-order concurrent imperative programming language, and recover an extension of the Aneris program logic [Krogh-Jespersen et al. 2020] for reasoning about distributed systems. The extension inherits from Aneris both (1) *horizontal modularity* via node-local and thread-local reasoning, which allows one to verify—and now refine—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. Using the Trillium instantiation, we extend these principles to history-sensitive refinement which additionally allows us to prove liveness properties of distributed systems. In §5 we consider an 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 a *fair* termination-preserving refinement of a suitable model. Figure 3 gives an overview of the different components and concepts developed in this work, how they depend on each other, and which formal system is used to reason about each part.

Contributions. In summary, we make the following contributions:

- We introduce Trillium, a language-agnostic separation logic framework for establishing history-sensitive refinement among traces of program executions and traces of abstract models through Hoare-style reasoning.
- We instantiate Trillium with AnerisLang to get an extension of Aneris that allows us prove refinements for distributed systems. We prove soundness of a wide collection of proof rules, including all the earlier proof rules of Aneris, with respect to the notion of Hoare triple provided by Trillium.
- We use this instantiation to prove the correctness of concrete implementations of two distributed protocols, two-phase commit and single-decree Paxos, by showing that they refine 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 an abstract TLA$^+$ specification. We also demonstrate how to use the *same* specification both to prove properties about the implementation, by relying on correctness theorems of the TLA$^+$ specification, and to show functional correctness of client programs.
- We further show functional correctness and strong eventual consistency of a concrete implementation of a Conflict-Free Replicated Data Type (CRDT). 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 non-distributed 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 model.

2 TRILLIUM: A TRACE PROGRAM LOGIC FRAMEWORK

The Trillium program logic is language agnostic and is defined with respect to an operational semantics given by a notion of expression $e \in \mathit{Expr}$, value $v \in \mathit{Val} \subseteq \mathit{Expr}$, evaluation context $K \in \mathit{Ectx}$, program state $\sigma \in \mathit{State}$ (a model of, *e.g.*, the heap and/or the network), and a *primitive reduction relation* $e_1, \sigma_1 \rightsquigarrow e_2, \sigma_2, \vec{e_f}$ that relates an expression e_1 and a state σ_1 to an expression e_2, a state σ_2, and a list $\vec{e_f}$ of expressions, corresponding to the threads forked by the reduction.

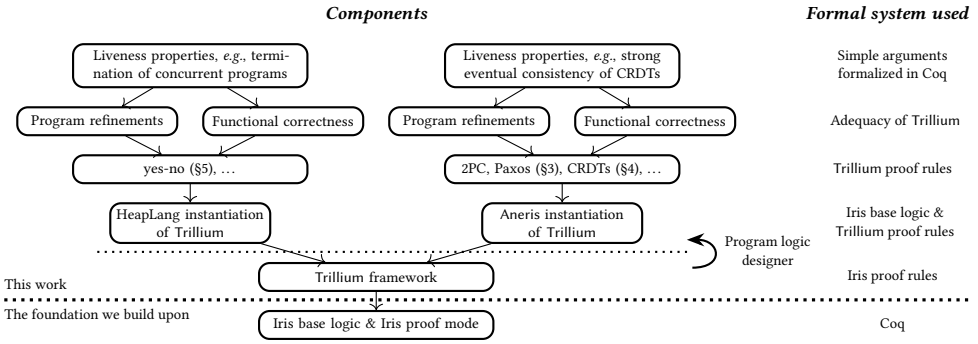


Fig. 3. An overview of the components described and developed in this work with arrows indicating dependency. The column to the right describes the formal system used to reason about the components in the row.

The top-level reduction relation $c \to c'$ on *system configurations* $c, c' \in Cfg = List(Expr) \times State$ is a standard interleaving small-step semantics using evaluation contexts, where the first component of a configuration is a list of expressions which denotes the threads of the system.

When concerned with program execution traces, we will only be interested in *valid traces*, written $valid(\tau)$, over configurations where each configuration reduces to the next according to the reduction relation of the operational semantics.

2.1 Program Logic

When instantiated with a programming language, the Trillium program logic comes with a set of low-level proof rules which relate the validity of some Hoare triples with the semantics of the language. When Trillium is instantiated with a concrete programming language, such as AnerisLang, the usual high-level proof rules are proved using the low-level rules. Once this work has been done, which happens only once for a given language, Trillium can be used like any Hoare-style logic.

A selection of the proof rules for the AnerisLang instantiation is shown in Figure 4; the notion of expression in Aneris also includes an ip address on which the expression is running. AnerisLang is a subset of the OCaml programming language with network primitives for creating and binding network sockets as well as sending (**`sendto`**) and receiving (**`receivefrom`**) messages. The operational semantics is designed so that the primitives closely model Unix sockets and UDP networking. A detailed description can be found in the supplementary material.

The proof rules include all of the earlier proof rules of Aneris. We write $\{P\}\ e\ \{v.Q\}^{\mathcal{M}}$ to explicate that the post condition is a predicate $\lambda v.\, Q$ on the result of evaluating e. Ht-frame and Ht-bind constitute the quintessential rules for modular reasoning in separation logic: the frame rule says that executing e for which we know $\{P\}\ e\ \{Q\}^{\mathcal{M}}$ cannot possibly affect parts of the heap that are separate from its *footprint* and the bind rule lifts a local specification to a more global specification in evaluation context K. The rules Ht-alloc and Ht-load are for reasoning about allocating and reading from references, respectively, on a node with ip-address *ip*. The rule Ht-sendto reasons about sending a message m over a socket z; both the pre- and postcondition contains Aneris-specific network resources concerned with the status of socket handles, which messages that have been sent (T) and received (R) at a particular address (a), and Aneris communication protocols (Φ). We refer to Krogh-Jespersen et al. [2020] for a detailed description of the Aneris resources and more proof rules.

$$\textsc{Ht-frame}\ \frac{\{P\}\,\langle ip;\, e\rangle\,\{Q\}^{\mathcal{M}}}{\{P * R\}\,\langle ip;\, e\rangle\,\{Q * R\}^{\mathcal{M}}} \qquad \textsc{Ht-bind}\ \frac{\{P\}\,\langle ip;\, e\rangle\,\{v.Q\}^{\mathcal{M}} \qquad \forall v.\{Q\}\,\langle ip;\, K[v]\rangle\,\{R\}^{\mathcal{M}}}{\{P\}\,\langle ip;\, K[e]\rangle\,\{R\}^{\mathcal{M}}}$$

$$\textsc{Ht-alloc}\ \{\mathsf{True}\}\,\langle ip;\, \mathbf{ref}\, v\rangle\,\{v.\ \exists \ell.\, v = \ell * \ell \mapsto_{ip} v\}^{\mathcal{M}} \qquad \textsc{Ht-load}\ \{\ell \mapsto_{ip} w\}\,\langle ip;\, !\,\ell\rangle\,\{v.v = w * \ell \mapsto_{ip} w\}^{\mathcal{M}}$$

$$\textsc{Ht-sendto}\ \{z \hookrightarrow_{ip} \mathsf{Some}(a) * a \rightsquigarrow (R, T) * to \Rrightarrow \Phi * \Phi(m, a, to) * a.\mathsf{ip} = ip\}\ \langle ip;\, \mathbf{sendto}\ z\ m\ to\rangle\ \{v.\ v = |to| * z \hookrightarrow_{ip} \mathsf{Some}(a) * a \rightsquigarrow (R, T \cup \{(m, a, to)\})\}^{\mathcal{M}}$$

Fig. 4. Selected proof rules of the Aneris instantiation of Trillium.

To reason about model refinement, Trillium admits *one* additional proof rule. For convenience, we show its AnerisLang instantiation Ht-take-step.

$$\textsc{Ht-take-step}\ \frac{\{P\}\,e\,\{Q\}^{\mathcal{M}} \qquad \delta \rightharpoonup_{\mathcal{M}} \delta' \qquad \mathrm{Atomic}(e) \qquad e \notin \mathit{Val}}{\{P * \mathsf{Model}_\circ(\delta)\}\,e\,\{Q * \mathsf{Model}_\circ(\delta')\}^{\mathcal{M}}}$$

The rule allows the state of the model to be updated during atomic operations by updating the $\mathsf{Model}_\circ(\delta)$ resource according to the stepping relation of the model. The rule can, naturally, be combined with other rules, such as Ht-inv-open which we show later in this section, to access the $\mathsf{Model}_\circ(\delta)$ resource from inside invariants. Note how the rule also *requires* the program to take a single step: it is atomic, so it takes *at most* a single step, and it is not a value, hence it *must* take a step. Symmetrically, the rule also prevents us from taking two consecutive steps in the model.

As a consequence of building on the Iris framework, the Trillium logic features all the usual connective and rules of high-order separation logic, some of which are summarized in Figure 5; the type of Iris propositions is *iProp*.[2]

$$\begin{aligned}
\sigma &::= 0 \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \mathit{Val} \mid \mathit{Expr} \mid \mathit{iProp} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \to \sigma \mid \ldots && \text{(Types)}\\
P, Q &::= x \mid \lambda x : \sigma.\, P \mid \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q && \text{(Propositional logic)}\\
&\mid \forall x : \sigma.\, P \mid \exists x : \sigma.\, P \mid t = u && \text{(Higher-order logic)}\\
&\mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v && \text{(Separation logic)}\\
&\mid \boxed{P} \mid \triangleright P \mid \Rrightarrow P \mid \boxed{a}^{\gamma} \mid \ldots && \text{(Iris connectives)}\\
&\mid \{P\}\,e\,\{v.Q\}^{\mathcal{M}} \mid \mathsf{Model}_\circ(\delta : \mathcal{M}) && \text{(Trillium connectives)}
\end{aligned}$$

Fig. 5. Syntax of Iris and Trillium. t and u represent arbitrary terms.

As as a separation logic, it has a separating conjunction $P * Q$ and the corresponding notion of implication, the *magic wand* $P \mathrel{-\!\!*} Q$, in that it satisfies *modus ponens*: $P * (P \mathrel{-\!\!*} Q)$ entails Q.

[2]To avoid the issue of reentrancy, invariants are annotated with a *namespace* and Hoare triples and update modalities with *masks*. We omit both for the sake of presentation in most of the paper as they are orthogonal to the novelties of Trillium.

The Iris base logic features invariants $\boxed{P}$ and user-defined ghost state $\boxed{a}^{\gamma}$. The logical support for both user-defined invariants and ghost state allows us to relate (ghost and physical) resources to each other. This is crucial for our specifications as already exemplified in the introduction.

The contents of invariants may be accessed in a carefully restricted way in Trillium using HT-INV-OPEN while verifying a program e. The rule permits us to access the contents of an invariant, which involves acquiring ownership of P before the verification of e and giving back ownership of P afterwards. Crucially, e is required to be *atomic*, meaning that the computation completes in a single step. Notice that invariants are just another kind of proposition and it can be used anywhere that normal propositions can be used, including invariants themselves, referred to as *impredicativity*. This is the reason P appears under a "later" modality $\triangleright P$; without, the rule for opening invariants is unsound. However, if one does not store invariants inside invariants or make similar impredicative constructions, one can generally ignore the later modality as we will do throughout the paper.

$$\frac{\text{HT-INV-OPEN}\quad \text{Atomic}(e) \qquad \{\triangleright P * Q_1\}\, e\, \{\triangleright P * Q_2\}^{\mathcal{M}}}{\left\{\boxed{P} * Q_1\right\} e \left\{\boxed{P} * Q_2\right\}^{\mathcal{M}}}$$

User-definable ghost state can be introduced via the proposition $\boxed{a}^{\gamma}$ which asserts ownership of a piece of ghost state a at ghost location γ. Resources can be *updated* through the *update modality* $\Rrightarrow P$. The update modality provides a way, inside the logic, to talk about the resources we *could own* after performing an update to what we *do own*. The intuition is that $\Rrightarrow P$ holds for a resource r, if from r we can do an update to some r' that satisfies P while not invalidating any existing resources. Using GHOST-UPDATE, we can update a user-defined ghost resource from $\boxed{a}^{\gamma}$ to $\boxed{b}^{\gamma}$ under the update modality if the particular user-defined ghost *resource algebra* permits it ($a \rightarrowtail b$). In a similar spirit, INV-ALLOC allocates an invariant $\boxed{P}$ by giving up ownership of P.

$$\frac{\text{GHOST-UPDATE}\quad a \rightarrowtail b}{\boxed{a}^{\gamma} \vdash \Rrightarrow \boxed{b}^{\gamma}} \qquad\qquad \text{INV-ALLOC}\quad P \vdash \Rrightarrow \boxed{P}$$

We refer to Jung et al. [2018] for a thorough treatment of how user-defined ghost state is constructed in Iris. In the remainder of this paper, we will simply describe user-defined ghost state in terms of the rules that govern the concrete instantiations that we use.

3 REFINEMENT OF TLA$^+$ SPECIFICATIONS

In the following, we discuss how to establish a 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 OCaml 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 Paxos can also be found in the official TLA$^+$-examples repository on GitHub. In our formalization, we have manually translated the TLA$^+$ system specifications into transition systems in Coq and proved their correctness properties. We argue that the translations are faithful and straightforward, however, it is not crucial for soundness that the translation is correct as the translated models have been proven correct independently in Coq.[3]

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

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 supplementary material. The development follows the same methodology as for the Paxos algorithm which we describe below.

3.1 Single-Decree Paxos

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

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

Model. The TLA$^+$ model of SDP is summarized in Figure 6. 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 values for any ballot less than b ($\mathit{HavePromised}(\mathcal{S}, Q, b)$) 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:

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

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

SDP-Phase1a
$$\frac{}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightarrow_{\mathsf{SDP}} \mathcal{S} \cup \{\mathsf{msg1a}(b)\}, \mathcal{B}, \mathcal{V}}$$

SDP-Phase1b
$$\frac{\mathsf{msg1a}(b) \in \mathcal{S} \qquad b > \mathcal{B}(a) \qquad \mathcal{V}(a) = o}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightarrow_{\mathsf{SDP}} \mathcal{S} \cup \{\mathsf{msg1b}(a, b, o)\}, \mathcal{B}[a \mapsto \mathsf{Some}(b)], \mathcal{V}}$$

SDP-Phase2a
$$\frac{\nexists v'.\, \mathsf{msg2a}(b, v') \in \mathcal{S} \qquad Quorum(Q) \qquad ShowsSafeAt(\mathcal{S}, Q, b, v)}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightarrow_{\mathsf{SDP}} \mathcal{S} \cup \{\mathsf{msg2a}(b, v)\}, \mathcal{B}, \mathcal{V}}$$

SDP-Phase2b
$$\frac{\mathsf{msg2a}(b, v) \in \mathcal{S} \qquad b \geq \mathcal{B}(a)}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightarrow_{\mathsf{SDP}} \mathcal{S} \cup \{\mathsf{msg2b}(a, b, v)\}, \mathcal{B}[a \mapsto \mathsf{Some}(b)], \mathcal{V}[a \mapsto \mathsf{Some}(b, v)]}$$

Fig. 6. TLA$^+$ specification of single-decree Paxos (SDP).

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

Theorem 3.1 (Consistency, SDP model). *Let* $\iota_{\mathsf{SDP}} = (\emptyset, \lambda\_.\, None, \lambda\_.\, None)$. *If* $\iota_{\mathsf{SDP}} \rightarrow^*_{\mathsf{SDP}} (\mathcal{S}, \mathcal{B}, \mathcal{V})$ *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 OCaml 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 supplementary material.

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

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.
```

the same ballot. In our Coq formalization, proposer p repeatedly issues new ballots of the form $k \cdot |\text{Proposer}| + p$ for $k \in \mathbb{N}$ 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 instantiate the Aneris logic with the model; the key part of the proof is to keep the $\text{Model}_\circ(\delta)$ resource in a global invariant that ties together the model state and the physical state with enough information to verify the implementation and for the refinement relation established through the adequacy theorem to be strong enough for proving our final correctness theorem (Corollary 3.2). Under this invariant we will *modularly* verify each Paxos role and each component in isolation.

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

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

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

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

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

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

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

Equipped with these resource we can state the invariant:

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

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

In the last part of the invariant, the $\mathit{BalCoh}(\mathcal{S})$ predicate simply requires that if $\mathsf{msg2a}(b, v) \in \mathcal{S}$ then $\mathsf{shot}(b, v)$ holds. This implies that by owning $\mathsf{pending}(b)$ you are the only entity that may propose a value for ballot b and it may never change. The $\mathit{MsgCoh}(\mathcal{S})$ predicate ties the physical state of the program to the model using Aneris-specific predicates for tracking the state of the network. This, for instance, forces acceptors and proposers to also add to the model state $\mathcal{S}$ any message they send over the network. Hence, to verify a proposer or an acceptor that sends a message, the proof must open the invariant, use Ht-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:

$$\{I_{\mathsf{SDP}} * \mathsf{MaxBal}_\circ(a, \mathsf{None}) * \mathsf{MaxVal}_\circ(a, \mathsf{None}) * \ldots\}\ \langle ip; \texttt{acceptor}\ L\ a\rangle\ \{\mathsf{False}\}$$

$$\{I_{\mathsf{SDP}} * \mathsf{pending}(b) * \ldots\}\ \langle ip; \texttt{proposer}\ A\ skt\ b\ v\rangle\ \{\mathsf{True}\}$$

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 can compose these specifications to get a single Hoare triple for a distributed system with both proposers, acceptors, and learners. By applying the adequacy theorem to this specification we get that the implementation indeed refines the TLA$^+$ model of SDP.[4]

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

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

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

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

we combine the adequacy theorem (Theorem 6.1) with our model correctness theorem (Theorem 3.1) to obtain the following corollary that *only* talks about the execution of the SDP implementation.

Corollary 3.2. *Let e be a distributed system obtained by composing n proposers, m acceptors, and k learners. For any T and σ, if $(e; \emptyset) \to^* (T; \sigma)$ and both $\mathit{ChosenI}(\mathit{messages}(\sigma), v_1)$ and $\mathit{ChosenI}(\mathit{messages}(\sigma), v_2)$ hold then $v_1 = v_2$.*

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

Functional Correctness of Clients. Corollary 3.2 is a meta-logic theorem (*e.g.*, in Coq) that only talks about the program execution and it follows as a consequence of the adequacy theorem and the model theorem. However, it is not only in the meta-logic that we can exploit properties of the model to prove properties about programs as the model is also embedded as a *resource* in the logic.

Listing 3 shows a client application that receives a message from two different Paxos learners and asserts that the two values are equal; if the two values do not agree, the program crashes. We can prove a specification for the client of the shape $\{I_{\mathsf{SDP}} * \ldots\}\ \langle ip;\ \texttt{client}\ a\rangle\ \{\ldots\}$. From the adequacy theorem it follows that the program is safe, *i.e.*, it does not crash, which means the asserted statement must always hold. In the proof of this specification, the client will receive ghost resources from the learners conveying that v_1 and v_2 have been chosen (*i.e.*, that a quorum of acceptors have accepted v_i). By opening the invariant I_{SDP} and hence obtaining the model resource $\mathsf{Model}_\circ(\mathcal{S}, \mathcal{B}, \mathcal{V})$, we can combine this knowledge with Theorem 3.1—a property exclusively of the model—and conclude that $v_1 = v_2$. Naturally, we may still compose a distributed system containing the client together with proposers, acceptors, and learner nodes and derive a specification for the full system. This single specification for the full distributed system entails *both* the refinement of the TLA$^+$ model and the safety of the programs running on all nodes.

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 (parts 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).[5]

Implementation. The implementation of the G-Counter in Figure 7 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.

In the code of the `install` function the `s` in `ref<s>` is the so-called *label* of allocation used to identify so-called allocation events. See the supplementary material for details of allocation events and how they allow us to state and prove properties regarding memory locations even before they are allocated. For instance, here we enforce that the state of a replica in the model should be zero before the local state of the replica is physically allocated and that it must match the state stored physically on the replica after its allocation.

[5]The full Coq proof amounts to about 2000 lines of proof scripts.

The state of a G-Counter replicas is a vector (an array), one element for each replica (including themselves), with the j^{th} element of the vector tracking the number of increments performed on the j^{th} replica. Note how the `incr` method on the i^{th} replica increments the i^{th} 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. 7. 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 cannot know the exact value of the counter; we only know that it is monotonically increasing. Figure 8 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 i^{th} 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).

$$\{\mathsf{gcounter}(i,k)\}\ \langle ip_i;\ \mathtt{query}()\rangle\ \{m.\ k \le m * \mathsf{gcounter}(i,m)\} \qquad \textsc{QuerySpec}$$
$$\{\mathsf{gcounter}(i,k)\}\ \langle ip_i;\ \mathtt{incr}()\rangle\ \{().\ \exists m.\ k < m * \mathsf{gcounter}(i,m)\} \qquad \textsc{IncrSpec}$$

Fig. 8. G-Counter query and increment node-local specification.

[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 is a free port.

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.

The eventual consistency property is a property about the entire execution of a program. Hence, we need to be able to formally define traces that capture the entire execution of a program, or its corresponding trace in the model, which can also include infinite executions. For this reason, we introduce possibly-infinite traces. These traces, like finite traces we have seen before, are sequences of elements with the difference that they can be finite, or infinite, or even empty. We will use possibly-infinite traces in conjunction with finite traces, often as a pair. The idea is that this captures the program execution (or a model trace) up to a certain point, the finite trace being the past while the possibly-infinite trace is the future. We denote possibly-infinite traces with letters with a dot on top, *i.e.*, $\dot{\tau}$ and $\dot{\kappa}$ for possibly-infinite program and model traces, respectively. We will also use similar notation to finite traces for operations on possibly-infinite traces, *e.g.*, we write $c :: \dot{\tau}$ for extending the possibly-infinite trace $\dot{\tau}$ with configuration c.

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. The refinement relation that we obtain between possibly-infinite traces of the program and the model allows us to show

- That any possibly-infinite model trace corresponding to a possibly-infinite execution trace that satisfies network fairness properties is model-fair.
- 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.

Putting all of the above together we can conclude that all program traces that satisfy network fairness properties are eventually consistent.

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 kinds of transitions (Figure 9) 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 i^{th} element of the i^{th} 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 i^{th} vector to be the result of merging (point-wise maximum) of the i^{th} 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 j^{th} replica in the past—the j^{th} replica could have been incremented after its state was sent over the network and before getting merged.

$$\frac{}{\delta \to_{\text{GC}} \delta[i \mapsto \delta_i[i \mapsto \delta_{i,i} + 1]]}\ \text{GC-IncrStep} \qquad \frac{\vec{v} \sqsubseteq \delta_j}{\delta \to_{\text{GC}} \delta[i \mapsto \delta_i \sqcup \vec{v}]}\ \text{GC-ApplyStep}$$

Fig. 9. 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 $\dot{t}$, we write $Unroll_n(t, \dot{t})$ for the finite trace obtained by taking the first n elements of $\dot{t}$ (if there are n elements, otherwise as many as available) and appending them on t. (Recall the intuition we gave when we introduced possibly-infinite traces. The $Unroll_n$ function simply computes the execution trace n steps into the future.) As an example, we have $Unroll_1(t, a :: \dot{t}) = t :: a$. We write $Drop_n$ for dropping the first n elements of a possibly-infinite trace. For example, we have $Drop_1(a :: \dot{t}) = \dot{t}$.

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

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

Note that here we write v_i for the i^{th} row of the matrix v and $v_{i,j}$ for the j^{th} component of the i^{th} vector. This definition simply states that for any replicas i and j, for any number of steps k, there is a k' such that 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 i^{th} replica is *eventually* merged into the j^{th} replica.We define eventual consistency, stability point, and convergence point as follows:

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

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 $\vec{v}$.

Theorem 4.1 (Model Eventual Consistency). *For all κ and $\dot{\kappa}$, if $ModelFair(\kappa, \dot{\kappa})$ then $ModelEvCons(\kappa, \dot{\kappa})$.*

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 i^{th} node as a client of the G-Counter—note how the label of the allocated location for the state of the i^{th} 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 8. We write c_n^{GC} for the initial configuration of the closed system of n replicas of G-Counter.

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 on the network. Since we are considering a concrete implementation here, we additionally assume some liveness properties of the implementation (including fairness of schedulers on 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:

$$NetFairSend(\tau, \dot{\tau}) \triangleq \forall i, j, n.\ \exists k.\ n \leq |\text{TraceSends}_{i,j}(Unroll_k(\tau, \dot{\tau}))|$$
$$NetFairRec(\tau, \dot{\tau}) \triangleq \forall i, n.\ \exists k.\ n \leq |\text{TraceRecs}_i(Unroll_k(\tau, \dot{\tau}))|$$
$$NetFairDel(\tau, \dot{\tau}) \triangleq \forall i, j, sev, k.\ sev \in \text{TraceSends}_{i,j}(Unroll_k(\tau, \dot{\tau})) \Rightarrow$$
$$\exists k', sev', rev.\ \text{same\_or\_happens\_after}(sev', sev) \wedge msg(sev') = msg(rev) \wedge$$
$$sev' \in \text{TraceSends}_{i,j}(Unroll_{k+k'}(\tau, \dot{\tau})) \wedge rev \in \text{TraceRecs}_j(Unroll_{k+k'}(\tau, \dot{\tau}))$$

$$NetFair(\tau, \dot{\tau}) \triangleq NetFairSend(\tau, \dot{\tau}) \land NetFairRec(\tau, \dot{\tau}) \land NetFairDel(\tau, \dot{\tau})$$

where $\text{TraceSends}_{i,j}$ is the list of all send events from the i^{th} replica to the j^{th} replica and 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 j^{th} 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 the beginning of execution the memory is not allocated. Hence, we follow an approach similar to the example in the supplementary material using allocation events. The eventual consistency theorem that we prove about our implementation is as follows:

Theorem 4.2 (Eventual Consistency). *Let $\dot{\tau}$ be a valid possibly-infinite trace starting from c_n^{GC} such that $NetFair(c_n^{GC}, \dot{\tau})$ holds. Then there exist $k \in \mathbb{N}$, $\dot{\kappa}$, and n locations $\ell_1, \ldots, \ell_n$ such that after k steps of computation (of the entire distributed system) all the locations storing local states of all replicas are allocated and these are exactly locations $\ell_1, \ldots, \ell_n$ (ℓ_i storing the state of the i^{th} replica). Furthermore, we have*

$$EvCons_{\ell_1,\ldots,\ell_n}\left(Unroll_k\left(c_n^{GC}, \dot{\tau}\right), Drop_k(\dot{\tau})\right)$$

where $EvCons_{\ell_1,\ldots,\ell_n}$ is the predicate stating that if there is a stability point (a point after which the j^{th} component of the state stored in ℓ_j does not change for any j) then there is a convergence point (a point after which the values stored in all locations is the same).

This theorem essentially says that there eventually is a point where all replicas have allocated their locations and that as of that point if there is a stability point, there must also be a convergence point.

Proof Sketch of Theorem 4.2. The proof is divided into two parts. We first show that a certain relation GcMainRel holds between the program trace and the model trace; this is essentially a simple consequence of the refinement relation that we have established, and which we obtain by the adequacy theorem. Afterwards, we prove that GcMainRel together with the network fairness properties implies *EvCons*.

We begin by showing that there exists $k \in \mathbb{N}$, $\dot{\kappa}$, and n locations $\ell_1, \ldots, \ell_n$ as described in the theorem and that the following holds:

$$\forall k'.\ \text{GcMainRel}\left((\ell_1, \ldots, \ell_n), Unroll_{k+k'}\left(c_n^{GC}, \dot{\tau}\right), Unroll_{k+k'}\left(\iota_n^{GC}, \dot{\kappa}\right)\right)$$

This simply states that GcMainRel holds for ever after the k^{th} step. Intuitively, the relation GcMainRel holds when:

(1) 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.
(2) 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.*
(3) At all times, the vectors stored on the heap and the model agree.

Note the subtlety of condition (1) 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

```
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 no b m

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

Fig. 10. The Yes and No threads.

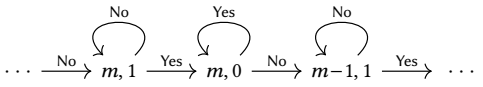


Fig. 11. The model $\mathcal{F}_{\text{YN}}$.

than the vector being sent as increment operations can take place in between the reading and the sending operations. Also, for condition (2), 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.

As for the second part of the proof, i.e., proving $EvCons_{\ell_1,\ldots,\ell_n}$, note how GcMainRel together with *NetFair* implies *ModelFair*. Assume that $\vec{v}$ is the current state of i^{th} 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 j^{th} 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:

$$\boxed{\exists locs, \delta.\ \text{Model}_\circ\,(\delta) * \mathit{HeapRel}(locs, \delta) * \mathit{SentRel}(locs, \delta) * \mathit{RecRel}(locs, \delta)}$$

Here the predicate *HeapRel* states that either the i^{th} replica has allocated its location and it stores exactly δ_i or δ_i is all 0's. See the example in the supplementary material for a detailed example where allocation events are used to establish a property similar to this property. 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 HT-TAKE-STEP to update the state of the model whenever the **cas** operations in the `perform_merge` or `incr` methods succeed—note that the **cas** operation is an atomic operation.

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.

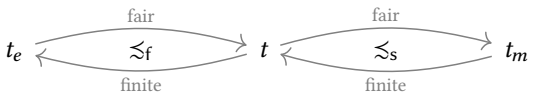
In a concurrent setting, the generally relevant notion of termination is *fair termination*, as most concurrent programs only terminate if the scheduler is fair. For example, the program presented in Figure 10, where two threads `yes` and `no` flip back and forth a shared Boolean `b`, does intuitively 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 $(T_1, \sigma_1) \xrightarrow{\text{tid}_1} (T_2, \sigma_2) \xrightarrow{\text{tid}_2} \cdots$, 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 by 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:

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

where 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 *roles* which act similarly to 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 11 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 $b = 0$, 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 $\precsim$ on their traces which entails that (1) holds, since fair termination is a trace property. The relation $\precsim$ 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 $\precsim_{\text{f}}$ and $\precsim_{\text{s}}$ on traces so that $t_e \precsim t_m$ iff there exists a trace t of an intermediate labeled STS $\text{Live}(\mathcal{F})$ such that the two following refinements hold:



(2)

The first relation $\precsim_{\text{f}}$ is defined as history-sensitive refinement of a model $\text{Live}(\mathcal{F})$ for a certain fixed relation ξ. The flexibility of Trillium's notion of refinement means that, by carefuly choosing the model and the relation ξ, it can be made to be a fairness-preserving termination-preserving refinement. A state of $\text{Live}(\mathcal{F})$ 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 $F(\rho) = \text{tid}$, can *increase* the fuel of ρ but must *decrease* the fuel of all the other roles $\rho' \in T^{-1}(\text{tid})$. 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 $\text{Live}(\mathcal{F})$ by ignoring stuttering steps. The relation $\precsim_{\text{s}}$ 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 $t_e \precsim_{\text{f}} t$ and if t_e is fair, then t is fair. Thus if 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 $t_e \precsim_{\text{f}} t$, which we do by proving a weakest precondition for the program e in Trillium instantiated with the model $\mathcal{M} \coloneqq \text{Live}(\mathcal{F})$. We use an instantiation 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 $\text{Live}(\mathcal{M})$: in particular, $\rho \mapsto_F f$ means that role ρ has fuel f, and $\text{tid} \mapsto_T R$ means that the thread tid is associated to the set R of roles. Finally, $\text{Model}_\circ(\delta)$ 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*

$$\left\{\mathsf{Model}_\circ\,(\delta_0) * 0 \mapsto_T R * \mathop{\Large *}_{\rho \in R} \rho \mapsto_F f_{init}\right\}\ e\ @\ 0\ \ \{0 \mapsto_T \emptyset\}^{\mathcal{F}}$$

holds for any f_{init}, R, 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 presented in the appendix 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 to the model $\mathcal{F}_{\text{YN}}$ depicted in Figure 11 two Booleans *ye* and *ne* to the states, where $ye = 0$ means Yes has finished, and $ne = 0$ means No has. The resulting model is explicated in the appendix.

We use the theorem above to prove that the program in Figure 10 is fairly terminating by establishing the weakest precondition, with $R := \{\text{Yes}, \text{No}\}$ and $\mathrm{f}_{init} := 30$. The proof of the weakest precondition is fairly simple and follows the methodology explained for the minimal example in the Introduction. See the supplementary material for complete definitions.

The approach presented here is similar in spirit to the one in the work of Tassarotti et al. [2017] but for reasoning about refinement of general concurrent programs with respect to abstract models. To the best of our knowledge, the expressiveness of the logics is roughly similar. The main difference is that Tassarotti et al. [2017] augments the Iris base logic with *linear* propositions, which requires modifying the definition of resource algebra to add a transition relation. We achieve similar results without heavy modifications, using that the authoritative state of the model is threaded through the weakest precondition, and by putting an exclusive structure on the set of roles owned by a thread, which prevents 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 THE SEMANTICS OF HOARE TRIPLES

In Trillium, Hoare triples are defined using more primitive notions as:

$$\{P\}\, e\, \{Q\}^{\mathcal{M}} \triangleq \Box(P \twoheadrightarrow \mathsf{wp}^{\mathcal{M}}\, e\, \{Q\})$$

where $\mathsf{wp}^{\mathcal{M}}\, e\, \{Q\}$ is the *weakest precondition* that is required for e to terminate safely in a value v satisfying Q *and* such that the execution of e refines the model $\mathcal{M}$. A technicality is that it is necessary to wrap the implication in a *persistence modality* $\Box$ to ensure that Hoare triples are duplicable and can be used repeatedly.

The Trillium weakest precondition 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]. The key idea and novelty of the Trillium weakest precondition is to track a model trace alongside a program execution trace and enforce that whenever the program takes a step according to the operational semantics, there is a state in the model that corresponds to the program step. Importantly, it does *not* mention an explicit model state or traces as the relationship between the program and the model trace is encapsulated inside the definition of the weakest precondition.

In order to avoid reentrancy issues, where invariants are opened in a nested (and unsound) fashion, Iris features *invariant namespaces* $\mathcal{N} \in \mathit{InvName}$ and *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$. Up until now, these matters have been omitted but to fully state and comprehend the definition of

the weakest precondition—and consequently our adequacy theorem—they are necessary evils. We emphasize that their use is entirely standard and identical to the use in the definition of the Iris weakest precondition.

The Iris base logic annotates each invariant $\boxed{P}^{\mathcal{N}}$ with a namespace $\mathcal{N}$ to identify the invariant and we annotate the weakest precondition with a mask $\mathcal{E}$ to keep track of which invariants are enabled and may be opened. In order to work with invariants formally in Iris, the update modality is annotated with two masks: ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$. We write $\Rightarrow_{\mathcal{E}}$ when $\mathcal{E}_1 = \mathcal{E}_2 = \mathcal{E}$ and $\Rightarrow$ when $\mathcal{E} = \top$, the set of all masks. As discussed earlier, the update modality is used to reason about ghost state akin to how a weakest precondition is used to reason about physical state. On top of this, the mask annotations $\mathcal{E}_1$ and $\mathcal{E}_2$ denote which invariants are enabled and may be opened before and after the modality is introduced, respectively. Intuitively, the proposition ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$ holds for resources that—given the invariants in $\mathcal{E}_1$ are enabled—can be updated to resources that satisfy P—with the invariants in $\mathcal{E}_2$ enabled—without violating the environment's ownership of resources. The contents of invariants may be accessed in a carefully restricted way (Inv-open-upd): to prove $\Rightarrow_{\mathcal{E}} Q$ we map open an invariant with namespace $\mathcal{N}$ and assume $\triangleright P$ as long as $\mathcal{N} \in \mathcal{E}$ and we can re-establish the invariant as well.

$$\textsc{Inv-open-upd}\quad \frac{\mathcal{N} \in \mathcal{E}}{\boxed{P}^{\mathcal{N}} * (\triangleright P \mathrel{-\!\!*} \Rightarrow_{\mathcal{E}\setminus\mathcal{N}} (\triangleright P * Q)) \vdash \Rightarrow_{\mathcal{E}} Q}$$

As we will see, the careful placement of the update modalities in the definition of the weakest precondition will require all invariants in the mask annotation $\mathcal{E}$ to be enabled after every physical step in the operational semantics, corresponding to the intuition that invariants can only be opened atomically. For more details on invariants and the update modality in Iris we refer to Birkedal and Bizjak [2017]; Jung et al. [2018].

As a consequence of the Trillium framework's generality and parameterization by both a language and a model, we have no knowledge of the physical state, model, and how they might be related and reflected in the logic. Therefore, we parameterize the weakest preconditions by a *trace interpretation* relation $S : \mathit{Trace}(\mathit{Cfg}) \times \mathit{Trace}(A_{\mathcal{M}}) \to \mathit{iProp}$ that ties program execution traces and model traces to Iris resources. In contrast, the standard Iris weakest precondition uses a state interpretation predicate $S : \mathit{State} \to \mathit{iProp}$ for reflecting the physical state, such as a heap, as resources in the logic to give meaning to, for example, the points-to connective $\ell \mapsto v$. The trace interpretation relation subsumes this notion and is able to reflect both the current physical state and model state, but also their histories and relation, as resources.

Given a *trace interpretation* S, the definition of the weakest precondition is defined by guarded recursion in the Iris base logic as follows (highlighting the novel aspects in blue):

$$\begin{aligned}
\mathsf{wp}^{\mathcal{M}}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq\ & \big(e \in \mathit{Val} * \Rightarrow_{\mathcal{E}} \Phi(e)\big) \vee \\
& \Big(e \notin \mathit{Val} * \forall \tau, \tau', \kappa, \sigma, K, T_1, T_2. \\
& \quad \mathit{valid}(\tau) * \tau = (\tau' :: (T_1 +\!\!+ K[e] +\!\!+ T_2, \sigma)) * S(\tau, \kappa) \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rightarrow^{\emptyset} \\
& \quad\ \mathit{reducible}(e, \sigma) * \\
& \quad\ \Big(\forall e_2, \sigma_2, \vec{e_f}.\, (e, \sigma) \rightsquigarrow (e_2, \sigma_2, \vec{e_f}) \mathrel{-\!\!*} \triangleright\, {}^{\emptyset}\!\Rightarrow^{\mathcal{E}} \\
& \quad\quad \exists \delta.\, S(\tau :: (T_1 +\!\!+ K[e_2] +\!\!+ T_2 +\!\!+ \vec{e_f}, \sigma'), \kappa :: \delta) * \\
& \quad\quad \mathsf{wp}^{\mathcal{M}}_{\mathcal{E}}\, e_2\, \{\Phi\} * \mathop{\Large *}_{e' \in \vec{e_f}} \mathsf{wp}^{\mathcal{M}}_{\mathcal{E}}\, e'\, \{\mathsf{True}\}\Big)\Big)
\end{aligned}$$

where $\mathcal{E}$ is a mask, $\mathcal{M}$ is a model, and $\Phi \in \mathit{Val} \to \mathit{iProp}$ a predicate on values. The definition is by case distinction. If the program has already terminated (*i.e.*, e is a value) the postcondition should hold. If the program is not a value, then for all model traces κ and valid execution traces τ where e is about to take a step on some thread *and* the trace interpretation holds, there are two requirements. First, the program should be *reducible*, which means it is able to take a thread-local step, ensuring safety. Second, with access to all the invariants in $\mathcal{E}$, for any possible configuration that e might step to, there should exist a model state δ such that the trace interpretation holds for the extended trace. Additionally, if the program makes a step, then the weakest precondition must hold for the reduced program as well as for all threads it might have forked-off.

In summary, the Trillium weakest precondition is a conservative generalization of the usual Iris-style weakest precondition that admits all the usual rules that you would expect from program logics built on top of Iris. In addition, it offers Hoare-style reasoning about the relationship between program execution traces and model traces through the addition of a single rule HT-TAKE-STEP.

6.1 Adequacy

Given a weakest precondition for a program e and binary relation ξ on execution traces and model traces, the adequacy theorem of Trillium concludes that e is a history-sensitive refinement of the model and that ξ holds for all possible traces generated by the program and the model. Importantly, the refinement property only relies on the definition of the operational semantics and the traces over model states: when a refinement property is established using Trillium, one does not need to trust Iris nor Trillium, but only that the operational semantics and the model are as intended.

The adequacy theorem shown below involves a technical condition that requires the ξ relation to be *finitary*: the set $\{\delta \mid \xi(\tau :: c, \kappa :: \delta)\}$ has to be finite, for any τ, κ, and c. This condition is necessary as the underlying model of the base logic of Iris is step-indexed over the natural numbers.[7] The property is generally straightforward to prove: often, as in all our examples, either the model itself is finitely branching or the program configuration determines exactly the current state of the model.

THEOREM 6.1 (ADEQUACY). *Let e be a program, σ_0 a program state, and Φ an Iris predicate on values. Let $\delta_0 \in \mathcal{M}$ be a model state and ξ a finitary binary relation on execution traces and traces of $\mathcal{M}$. If*

$$\models\!\!\Rightarrow_{\top} S((e, \sigma_0), \delta_0) * \mathit{wp}^{\mathcal{M}}_{\top}\, e\, \{\Phi\} * \mathit{AlwaysHolds}(\xi, e, \sigma_0, \delta_0)$$

then $(e, \sigma_0) \precsim_{\xi} \delta_0$ holds in the meta-logic. Here $\mathit{AlwaysHolds}(\xi, e, \sigma, \delta)$ is the Trillium predicate

$$\forall \tau, \kappa. \begin{pmatrix} S(\tau, \kappa) * \mathit{valid}(\tau) * |\tau| = |\kappa| * \mathit{first}(\tau) = (e, \sigma_0) * \mathit{first}(\kappa) = \delta_0 * \\ \left(\forall \tau', \kappa', c', \delta'.\ \tau = \tau' :: c' \wedge \kappa = \kappa' :: \delta' \Rightarrow \xi(\tau', \kappa')\right) * \\ \begin{pmatrix} \forall e_1, \ldots, e_n, \sigma.\ \mathit{last}(\tau) = (e_1, \ldots, e_n; \sigma) \Rightarrow \\ (\forall 1 \le i \le n.\ e_i \in \mathit{Val} \vee \mathit{reducible}(e_i, \sigma)) \wedge (e_1 \in \mathit{Val} \Rightarrow \Phi(e_1)) \end{pmatrix} \end{pmatrix} \twoheadrightarrow {}^{\top}\!\!\models\!\!\Rightarrow^{\emptyset} \xi(\tau, \kappa)$$

Intuitively, the *AlwaysHolds* predicate simply says that ξ should follow from all invariants, *cf.*, the update modality masks. When proving this, one is additionally allowed to assume that the trace interpretation S holds, the execution trace is valid and the trace starts from e, σ_0, and δ_0; ξ holds for the prefixes of the traces; none of the threads at the current execution point are stuck; and if the first thread (corresponding to e) has evaluated to a value, then the postcondition Φ holds.

[7] The finiteness condition does not restrict the properties we can transport along a refinement, see §4 and §5. 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, it lacks commutation rules for the later modality.

Refinements for Infinite Executions. The adequacy theorem concludes a refinement relation for finite program executions, however, it straightforwardly extends to infinite executions as well. A *possibly-infinite trace* (over some set) is a finite or infinite sequence (of elements from the set). For possibly-infinite traces, refinement is a predicate on pairs of traces: a finite trace, corresponding, intuitively, to the trace up until now and a possibly-infinite trace corresponding to the remaining execution. Let τ and κ denote finite traces and $\dot{\tau}$ and $\dot{\kappa}$ possibly-infinite traces. We define history-sensitive refinement between possibly-infinite execution traces and model traces coinductively as follows:

$$(\tau, \dot{\tau}) \mathrel{\dot{\precsim}_\xi} (\kappa, \dot{\kappa}) \triangleq \begin{cases} \xi(\tau, \kappa) & \text{if } \dot{\tau} = \emptyset \text{ and } \dot{\kappa} = \emptyset \\ \xi(\tau, \kappa) \wedge (\tau :: c, \dot{\tau}') \mathrel{\dot{\precsim}_\xi} (\kappa :: \delta, \dot{\kappa}') & \text{if } \dot{\tau} = c :: \dot{\tau}' \text{ and } \dot{\kappa} = \delta :: \dot{\kappa}' \end{cases}$$

Intuitively, this definition states that all finite prefixes of the possibly-infinite traces preserve ξ.

Corollary 6.2 (Possibly-Infinite Refinement). *Let τ, κ be finite traces such that $\tau \precsim_\xi \kappa$, and let $\dot{\tau}$ be a possibly-infinite program trace such that τ and $\dot{\tau}$ are valid. Then there exists a possibly-infinite model trace $\dot{\kappa}$ such that $(\tau, \dot{\tau}) \mathrel{\dot{\precsim}_\xi} (\kappa, \dot{\kappa})$ holds.*

Note that it is *not* be possible to extract history-sensitive refinement relations from the usual weakest precondition in an Iris-style program logic. To see why, consider the example from the introduction. Using Iris's mechanism for user-defined ghost state, one can easily definite ghost state modelling a monotonically increasing counter, *e.g.*, a resource $\boxed{n}^\gamma$ such that

$$\boxed{n}^\gamma \vdash \Rrightarrow \boxed{n+1}^\gamma$$

is the only way it can be updated. Using this ghost theory, one *could* also prove a similarly-looking specification of the shape

$$\left\{\boxed{\exists n.\, \ell \mapsto n * \boxed{n}^\gamma}\right\} \mathsf{inc} \{\ldots\}$$

using the standard weakest precondition theory from Iris. However, this specification would *also* be satisfied by, for example, an implementation with threads that increment the counter by two, which clearly does not satisfy our refinement notion. If the counter represents a digital clock, for example, it would be far from ideal if it was allowed to skip every other minute.

The fundamental problem is that resource updates are *transitive* which means that if a particular resource algebra allows a resource update from a to b and from b to c it will also allow it to be updated directly from a to c; in particular, a ghost theory that allows our counter to progress as 0-1-2-... will also allow it to progress directly from 0 to 2. In Trillium, the model is encapsulated in the weakest precondition whose definition forces us to match up exactly *one* model step per computation step. In the Aneris instantiation (as detailed in the next section) we consider the reflexive closure, however in §5 we do not, which is crucial for the soundness of our method for showing fair termination-preserving refinement.

6.2 Aneris Instantiation of Trillium

To instantiate Trillium with AnerisLang, we define the trace interpretation using three components:

$$S(\tau, \kappa) \triangleq \mathsf{physSI}(\mathit{last}(\tau)) * \mathsf{Model}_\bullet(\mathit{last}(\kappa)) * \mathit{stuttering}(\kappa)$$

The physSI predicate corresponds to a state interpretation and associates the physical Aneris state, *i.e.*, the heap and the network, to Aneris resources, akin to how the heap is associated to the $\ell \mapsto v$ resource in standard Iris instantiations. The $\mathsf{Model}_\bullet$ predicate ties the *current* state of the model to an instance of the *authoritative resource algebra* [Jung et al. 2018]. The $\mathsf{Model}_\bullet$ predicate (the

authoritative part of the model) comes with a counterpart $\mathsf{Model}_\circ$ (the fragmental part of the model) satisfying

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

As discussed in the introduction, this gives the user a way of connecting the model's current state to other separation logic resources through the $\mathsf{Model}_\circ(\delta)$ resource. The *stuttering* predicate is defined as

$$\mathit{stuttering}(\kappa) = \begin{cases} \mathit{last}(\kappa') = \delta \lor \mathit{last}(\kappa') \rightharpoonup_{\mathcal{M}} \delta & \text{if } \kappa = \kappa' :: \delta \\ \mathsf{True} & \text{otherwise} \end{cases}$$

which allows the model trace to *stutter*: a model state in the trace is either the same as the previous or it is related to the previous state by a *single step* of the transition relation. When refining implementations of distributed systems, it is natural to allow stuttering on the model side given that we wish to relate a detailed program execution to a more abstract model.

7 RELATED WORK

We discuss some further related work not already discussed throughout 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.

Refinement in Iris. There has been earlier work on proving refinements using Iris. Most of this work, however, has focused on *contextual* refinement, where a (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 [Gäher et al. 2022; Spies et al. 2021; Tassarotti et al. 2017]. Perennial [Chajed et al. 2019] defines correctness of a system using *concurrent recovery refinement*, requiring that the (possibly crashing) implementation and specification STS has the same external I/O. This notion of refinement is much coarser and does not allow you to prove, *e.g.*,

fair termination. Tassarotti and Harper [2019] relates concurrent probabilistic programs to abstract specifications denoting indexed valuations, exhibiting a probabilistic coupling when assuming that the implementation terminates.

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.

8 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

Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. 247–256. https://doi.org/10.1109/LICS.2001.932501

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

Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Log. http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf. (2017).

Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. 2012. A Concurrent Logical Relation. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 107–121. https://doi.org/10.4230/LIPIcs.CSL.2012.107

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

Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. 2022. Applying formal verification to microkernel IPC at meta. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*. 116–129. https://doi.org/10.1145/3497775.3503681

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. 243–258. https://doi.org/10.1145/3341301.3359632

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

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

Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy game: verifying a local generic solver in Iris. *Proc. ACM Program. Lang.* 4, POPL (2020), 33:1–33:28. https://doi.org/10.1145/3371101

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

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM*

Program. Lang. 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. 2018. Paxos Consensus, Deconstructed and Abstracted. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 912–939. https://doi.org/10.1007/978-3-319-89884-1_32

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

Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed causal memory: modular specification and verification in higher-order distributed separation logic. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434323

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

Klaus Havelund, Michael R. Lowry, and John Penix. 2001. Formal Analysis of a Space-Craft Controller Using SPIN. *IEEE Trans. Software Eng.* 27, 8 (2001), 749–765. https://doi.org/10.1109/32.940728

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

Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

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, 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

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017c. 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*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.3009855

Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. 2020. Verifying concurrent search structure templates. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 181–196. https://doi.org/10.1145/3385412.3386029

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

Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473571

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, Vol. 12075)*, Peter Müller (Ed.). 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

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

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, Vol. 9032)*, Jan Vitek (Ed.). 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, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

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

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013a. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. *SIGPLAN Not.* 48, 9 (sep 2013), 377–390. https://doi.org/10.1145/2544174.2500600

Aaron J. Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013b. Logical Relations for Fine-Grained Concurrency. *SIGPLAN Not.* 48, 1 (jan 2013), 343–356. https://doi.org/10.1145/2480359.2429111

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

Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*. 100–115. https://doi.org/10.1145/3497775.3503689

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

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, Vol. 8461)*, Erika Ábrahám and Catuscia Palamidessi (Eds.). Springer, 33–48. https://doi.org/10.1007/978-3-662-43613-4_3