Modal Assertions for Actor Correctness

Colin S. Gordon
Drexel University
csgordon@drexel.edu

Abstract

The actor model is a well-established way to approach to modularly designing and implementing concurrent and/or distributed systems, seeing increasing adoption in industry. But deductive verification tailored to actor programs remains underexplored; general concurrent logics could be used, but the logics are complex and full of features to reason about behaviors the actor model strives to avoid.

We explore a relatively lightweight approach of extending a system for proving sequential program correctness with means to prove safety properties of actor programs (currently, assuming no faults). We borrow ideas from hybrid logic, a modal logic for stating assertions are true at a particular point in a model (in this case, a particular actor's local state). To make such assertions useful, we stabilize them using rely-guarantee-style reasoning over local actor states, and only permit sending stable versions of these assertions to other actors. By carefully restricting the formation of assertions that a proposition is true at a certain actor, we avoid the need for actors to handle each others' rely-guarantee relations explicitly. Finally, we argue that the approach requires only modest adjustments beyond applying traditional sequential techniques to actors with immutable messages, by implementing most of the logic as a Dafny library.

CCS Concepts • **Theory of computation → Modal and temporal logics**; **Program specifications**; • **Computing methodologies → Concurrent computing methodologies**.

Keywords Actors, Rely-Guarantee, Modal logic

ACM Reference Format:
Colin S. Gordon. 2019. Modal Assertions for Actor Correctness. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358499.3361221

AGERE '19, October 22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6982-4/19/10…$15.00
https://doi.org/10.1145/3358499.3361221

1 Introduction

The actor model [23] is a well-established approach to structuring concurrent or distributed programs, addressing the most prominent challenge of concurrent shared-memory programming with threads (i.e., data races) by completely forbidding shared mutable state, instead requiring actor processes to exchange immutable messages to update exclusively-actor-local mutable state. Data race freedom [8, 21] for actors removes some of the most brittle concurrency bugs and makes it sound to use purely sequential reasoning techniques to verify properties of an actor's local behavior. Unfortunately reasoning about a single actor's behavior at a time is often insufficient. One actor's correctness may depend on knowing information about *other* actors, such as in consensus algorithms, where an operation is committed if a majority of nodes have agreed to it — so specifications must refer to other nodes' states, and proofs must ensure other nodes preserve truth of the shared information.

This idea of one part of a program interfering with the *proof assumptions* of another part is how Owicki and Gries [36] approached verification of shared memory concurrent programs using threads. This attacks the essence of how concurrency complicates program reasoning, but requires checking that *every* operation in *every* thread preserves the truth of *every* assertion in *every* other thread's proof. Jones [26] proposed *rely-guarantee reasoning* to simplify this: summarizing for each thread (1) a *guarantee* to other threads of the system that its interference on global state would not exceed a certain threshold (given as a binary relation on the state before and after each statement in the thread), and (2) a *rely* relation stating an upper bound on what that thread's proof assumed other threads might do. Then each thread's proof was conducted using only (3) *stable* assertions (those whose truth was preserved by any action whose specification fell within the rely relation) and (4) when threads were composed in parallel they were checked for *compatibility*: that each thread's guarantee was a subrelation of the other's rely relation, ensuring the assumptions each thread made about the other's behavior were sound. Rely-guarantee style reasoning has since been integrated into various flavors of separation logic [13, 15, 44], and become an implicit reasoning principle underlying a variety of newer concurrent program logic constructs [12, 27, 35, 39]. Gordon et al. [19, 20] and Militão et al. [33, 34] even adapted rely-guarantee reasoning to treat interference between aliases, regardless of whether the interference was concurrent or not. These ideas could be applied to actor programs, but this is a heavyweight approach: these techniques include rich support for varieties of interference that actor systems are designed to *avoid by construction*. For programs designed to suit the strengths of the actor model, it would be appealing to have a *lightweight* way to extend the local sequential reasoning supported by data-race-free actors to permit useful reasoning about other actors.

We give a lightweight adaptation of rely-guarantee-style reasoning to actors, borrowing ideas from hybrid logic. We extend a sequential program logic with an assertion $@_i(P)$, stating that P is true of the state at actor i. Thus one actor's verification assumptions may refer to facts about another actor's state, but at runtime an actor's state remains accessible only to the actor itself. Of course, this alone might permit an actor to assume another is in exactly a specific state even when the actor might change its state, so additional constraints are required. We ensure such assertions are stable (in the rely-guarantee sense) by equipping each actor with a guarantee relation describing how it may update its own state upon processing a message. Assertions about an actor's state are required to be stable with respect to its guarantee, and this is enforced at the time such a property is established: only actor i may initially prove a proposition of the form $@_i(P)$. To support our "lightweight" claim, we also show that assuming reference immutability [8, 21], a sequential verification system (i.e., Dafny [28]) can provide these principles mostly as a library.

2 A Motivating Example

Let us consider a simple actor program to motivate some informal reasoning; later we verify the example, but for now we describe it only in prose (Figure 6 shows code). Let us assume a model similar to that used by Akka [30]. In Akka, actors are implemented as a JVM class with a single message handling method that handles all incoming messages. Actors are referred to using ActorRefs, which are essentially handles to specific actors in place of direct object references. Each actor has a single mailbox, and the actor system itself is responsible for invoking an actor's message handling method once for each message received. Sending messages is asynchronous: the message handler can send many messages, but send is non-blocking and no success or failure indication is provided; the only way for an actor to know another actor received and processed a message is to later receive a response message, in a later invocation of the message handler.

The system we are interested in verifying consists of two actors. The first is a simple counter actor: it keeps a counter locally, accepts messages indicating the actor should increment by a certain amount, and replies with the new value. The second is a "manager" of sorts, which acts as a sort of proxy to the counter: it can forward increment requests from external clients (for simplicity assume it does not forward along the counter's reply), and it can respond to client requests for a lower bound on the counter's value. The value it provides is in caching a lower bound on the counter locally. If a client requests a lower bound it can reply immediately. If a client requests an increment, the manager forwards it along and will later receive a reply from the counter with a newer value, which it can use to update its lower bound.

Let us consider an informal argument we might use to convince ourselves the manager and counter are correct. There is one key system invariant: the manager's local cached value is always a lower bound on the actual counter's value. As long as this is true, it will always be correct for the manager to reply with its latest cached value. Ensuring this is true requires a two-state invariant [29] on the messages from the counter to the manager: that the value contained in the message *is and remains* less than or equal to the current value. Assuming immutable messages, the only way this could be violated is if the counter might decrement its local value. This will not occur for reasonable implementations of an increment-only counter. So as long as every value sent to the manager is already no larger than its local count (trivially true if it simply always sends exactly its current count at that time), this two-state invariant [29] — an invariant on how any two successive states are related — holds. This means every time the manager receives an update from the counter, it can safely update its local cached copy and preserve the invariant.

Making this informal argument formal requires supporting a few key styles of reasoning:

- The invariant of the manager must be able to mention state of the counter
- Some part of the proof must be able to check that the manager's invariant is *stable* with respect to the behavior of the counter
- Some part of the proof must check that the counter only sends true lower bounds
- Messages from the counter with lower bounds must also communicate the lower bound property

This actually permits a wide range of formalizations. Ideally, though, the local proof of the manager's code should not concern itself directly with the details of the counter's local behavior: the only things the manager's proof *must* know locally are (1) the lower bound and (2) the fact that the lower bound remains a lower bound (it does not necessarily need to know *why*). An intuitive adaptation of classic rely-guarantee techniques would require the manager's proof to explicitly contain a bound on the counter's behavior and check stability of the lower bound assertion. Alternatively, an intuitive adaptation of something like rely-guarantee references [19, 20] to actor references would do the same, exposing summaries of the counter's possible state changes to the manager. But Vafeiadis [43] showed there are a range of possible ways to organize stability checks in rely-guarantee-style systems. So we would prefer to shift the burden of stability checks — and therefore, *all* explicit knowledge of the counter's behaviors — to the proofs of the counter itself.

We would like to take the following high-level approach:

1. Extend the assertion language over local state with a way to talk about an assertion true at another actor
2. Equip each actor with a binary relation that upper-bounds how its receive method changes its *local* state
3. Require any proofs that something is true at a particular actor to originate with that actor
4. Allow actors to attach logical claims to messages
5. Require actors to "promise" any logical claim sent in a message will be upheld

3 Hybrid Logic for Actors

Modal logics are logics that study some form of *contingent* truth, with operators reflecting that something may not currently be true, but may be true in a different circumstance, time, or place. The classic example is the modal logic of necessity, where $\Box P$ means P is *necessarily* true. Modal logics play an outsized role in program verification, because execution of program fragments corresponds to constructing alternative situations, in which different claims about program state may be true. This includes temporal logic [37], as well as standard program logics. Dynamic logic [16, 22, 38] includes a modality *indexed by programs*: $[\alpha](P)$ is the statement that P is true after executing program α — it is true in exactly those states where executing α will make P true, making it equivalent to the weakest precondition [11] of α with respect to P. This can be exploited for verification directly [1], or used to recover Hoare triples: $\{P\}\ C\ \{Q\}$ is representable in dynamic logic as $P \to [C](Q)$ — that the precondition P implies that after executing C, Q will be true. This is how Iris [27] derives triples.

A class of modal logics that has not, to the best of our knowledge, been exploited in verification is that of *hybrid* logic [6, 17, 18]. Hybrid logics extends a modal logic's language of propositions with two key ideas. *Nominals* $\iota \in \mathcal{N}$ uniquely identify points in a model (e.g., a particular state), so there is exactly one point in the model where a nominal ι is true. *Satisfaction operators* are modal operators indexed by nominals, which enable claims about the truth of another proposition at some arbitrary point in the model identified by a nominal: $@_\iota(P)$ asserts that P is true in the (unique) state identified by the nominal ι.

This style of reasoning seems well-suited to reasoning about actors: nominals correspond to the existing notion of a reference to a specific actor, so $@_\iota(P)$ would then represent the assertion that P was true of the local state of actor ι. This section outlines an approach to making this idea useful for verifying actor programs subject to some simplifying assumptions. So for example, the invariant of the manager from Section 2 could be characterized as $\exists v.\, b = v \land @_c(v \leq \mathit{count})$, assuming local variable b at the manager holds logical value v (valid in both actors' states), and at the counter (nominal c) this logical value is a lower bound.

Of course, hybrid logic's satisfaction operators by themselves assume all possible states are named by nominals, which would correspond to a single global program state with many actors. To reason about actor *programs* we must combine this with the ability to model program changes — in this case, dynamic logic. This leads to model / program states consisting of sets of individual actor states, where the truth of a proposition depends on both the general program state and (informally) a choice of which actor's point of view to adopt when interpreting propositions — so an actor's code will be verified from the "point of view" of that actor. From a modal logic perspective, this makes our endeavor a kind of 2-dimensional modal logic [40], with different modalities acting on different aspects of states.

3.1 A Multi-Dimensional Multi-Modal Model

We will give a Kripke model for a combined dynamic logic of actors' message handlers with hybrid logic to model each others' state. In modal logics, Kripke models are commonly used to give semantics to a logic. They consist of a set W of *worlds* and a family of binary relations on worlds describing their relationships. Worlds are intuitively the set of "situations" in which the truth of a formula may be considered — in our case, program states. The relations are used to give semantics to modal assertions that relate different states. In dynamic logics like ours, these correspond to programs that may modify program state.

We assume a universe $\mathcal{R}$ of actor *references* as nominals. We assume a basic propositional dynamic logic for purely-local actions (i.e., no send primitives) over a local state $\mathsf{LState} = \mathsf{Var} \rightharpoonup \mathsf{Nat} \cup \mathcal{R}$, whose commands are drawn from $\alpha \in \mathsf{Primitive}$, with (possibly-non-deterministic) command semantics given as $[\![-]\!] : \mathsf{Primitive} \to \mathsf{BinRel}(\mathsf{LState})$. In our examples we assume this set of primitives includes assignment between variables and basic arithmetic expressions.

We construct our models $\mathcal{M} = \langle W, R_{C \in \mathsf{Command}} \rangle$ according to Figure 1. A world (program state) is an $\mathcal{R}$-indexed finite set of actor states. An actor state is a triple of a local state (LState as above), a set of messages the actor has sent (message type, destination, and value), and a binary relation giving an upper bound on any local behavior they might have — the guarantee relation of each actor. Our semantics will only collect messages sent: in distributed settings, networks may reorder, duplicate, or drop messages, so we only enforce that if a message is delivered, then it was previously sent.

The transition relation is indexed by a particular command and a particular actor: $R_C(\iota)$ relates pre- and post-states of a particular actor referenced by ι executing command C. This updates the state of the actor in question differently than those of other possible actors: the state of the actor that is assumed to execute C is updated in accordance with C's local semantics. Each other actor's state is updated in *some* way corresponding to the reflexive transitive closures of its guarantee relation, possibly also sending messages. We

$$W = \mathcal{R} \rightharpoonup ((\mathsf{LState} \times \mathsf{set}\ (\mathsf{Token} \times \mathcal{R} \times (\mathsf{Nat} \cup \mathcal{R})\})) \times \mathsf{BinRel}(\mathsf{LState}))$$

$$R_C(\iota) = \left\{ (W, W') \;\middle|\; \begin{array}{l} \mathsf{dom}(W) = \mathsf{dom}(W') \wedge \\ (\forall \iota', s, g.\, W(\iota') = (s, g) \Rightarrow \exists s'.\, W'(\iota') = (s', g) \wedge (\mathsf{if}\ (\iota = \iota')\ \mathsf{then}\ (s, s') \in [\![C]\!]\ \mathsf{else}\ (s, s') \in (g^* \times \subseteq)) \end{array} \right\}$$

$$[\![\mathsf{send}(t, x, y)]\!](l, m) = (l, m \cup (t, l(x), l(y))) \qquad [\![C; C']\!] = \{(a, c) \mid \exists b.\, (a, b) \in [\![C]\!] \wedge (b, c) \in [\![C']\!]\}$$

Figure 1. Kripke model for an actor-based hybrid modal logic.

$$\begin{array}{l}
\mathcal{M}, w, \iota \vDash \mathsf{true}\ \text{always} \\
\mathcal{M}, w, \iota \vDash P \vee Q \Leftrightarrow \mathcal{M}, w, \iota \vDash P \text{ or } \mathcal{M}, w, \iota \vDash P \\
\mathcal{M}, w, \iota \vDash P \wedge Q \Leftrightarrow \mathcal{M}, w, \iota \vDash P \text{ and } \mathcal{M}, w, \iota \vDash P \\
\mathcal{M}, w, \iota \vDash \neg P \Leftrightarrow \mathcal{M}, w, \iota \nvDash P \\
\mathcal{M}, w, \iota \vDash \mathsf{ActorRef}(x) \Leftrightarrow w(x) \in (\mathcal{R} \cap \mathsf{dom}(w)) \\
\mathcal{M}, w, \iota \vDash \iota' \Leftrightarrow \iota = \iota' \\
\mathcal{M}, w, \iota \vDash @_{\iota'}(P) \Leftrightarrow \mathsf{stable}(w, \iota', P) \wedge \mathcal{M}, w, \iota' \vDash P \\
\mathcal{M}, w, \iota \vDash \mathsf{Guar}(g) \Leftrightarrow \exists s, g'.\, w(\iota) = (s, g') \wedge g \subseteq g' \\
\mathcal{M}, w, \iota \vDash [C](P) \Leftrightarrow \forall w' \in R_C(\iota).\, \mathcal{M}, w', \iota \vDash P \\
\mathcal{M}, w, \iota \vDash \exists x.\, P \Leftrightarrow \text{for some } v, \mathcal{M}, w, \iota \vDash P[x \mapsto v] \\
\mathcal{M}, w, \iota \vDash \forall x.\, P \Leftrightarrow \text{for any } v, \mathcal{M}, w, \iota \vDash P[x \mapsto v] \\
\hline
\mathcal{M}, w, \iota \vDash x \leq n \Leftrightarrow w(\iota)(x) \leq n \\
\mathcal{M}, w, \iota \vDash x = v \Leftrightarrow w(\iota)(x) = v \\
\mathcal{M}, w, \iota \vDash x \leq y \Leftrightarrow w(\iota)(x) \leq w(\iota)(y) \\
\mathcal{M}, w, \iota \vDash x = y \Leftrightarrow w(\iota)(x) = w(\iota)(y) \\
\mathcal{M}, w, \iota \vDash x = y \oplus v \Leftrightarrow w(\iota)(x) = w(\iota)(y) \oplus v \\
\text{where } \mathsf{stable}(w, \iota, P) = \\
\quad \forall w'.\, w' = w[\iota \mapsto (s', g) \mid s(\iota) = (s, g) \wedge (s, s') \in g] \Rightarrow \\
\qquad \mathcal{M}, w', \iota \vDash P
\end{array}$$

Figure 2. Semantics of assertions

build the full programming language by specifying the set $C \in \mathsf{Command} ::= \alpha \mid \mathsf{send}(t, x, y) \mid C; C$. Sends accept a Token indicating the message type (used to select which invariants are intended to hold at the sender, in terms of the value sent), and their semantics add the message to the set of messages sent by the current actor. We lift the semantics of primitives to operate on a pair of LState and message sets. Sequential composition's semantics is the relational composition of the nested command semantics. For brevity we omit loops and assume primitives include conditional updates (guarded [11] primitives).

This allows us to define the semantics of assertions in our language, relative to both a global state *and* a particular actor; Figure 2 defines the relation $\mathcal{M}, w, \iota \vDash P$ which is read as "in model $\mathcal{M}$, when considering truth from global state $w \in W$ and actor ι, P is considered true." Standard assertions are defined in the standard way (e.g., $P \vee Q$ is true if either P or Q is true). We include an assertion that a certain value is a valid actor reference. Actor references may be used as assertions, asserting that the code is running in the named actor (the standard interpretation of nominals).

The key case in the semantics is the interpretation of $@_\iota(P)$. This requires, as suggested in Section 2, that P be true from the viewpoint of actor ι (note the change in actor reference when checking P in that case). It also requires P to be stable. This stability check has the same intuitive meaning as in traditional rely-guarantee reasoning (that P's truth is preserved by changes within an upper bound), but notice that this check occurs *in the semantics of assertions* rather than in the proof theory of the logic. This means that proofs (1) do not need to concern themselves with stability checks for other actors' satisfaction assertions, and (2) do not even need to know what other actors' guarantees are!

From the perspective of the model, the assertion carries its own stability proof with it. Of course, stability must still be proven somewhere (in the rule for *introducing* a satisfaction assertion). The model's stability check has additional subtlety. Notice that it only checks stability (directly) with respect to changing the single actor's local state: the check is that the actor where P is true cannot perform any *local* action that invalidates P. Critically, this restriction allows us to give rules for introducing a satisfaction assertion that are also local to a single actor. If P contains further satisfaction operators referring to truth at *other* actors, those assertions' stability is already guaranteed separately, and these nested assertions' stability proofs for other actors' assertions can be combined with that for P's stability at ι to show all actors preserve P.

The assertion $\mathsf{Guar}(g)$ asserts that g under-approximates the local actor's guarantee. This is used in every proof rule for every primitive: in addition to handling the logical consequences of each primitive on what assertions are true, each action must fall within the guarantee.

The class of assertions $[C](P)$ are standard for dynamic logic, adapted to our non-standard model: it asserts that P should be true — at the same actor — after executing command C. We include universal and existential quantification, though we assume quantified variables are distinct from program variables. Finally we assume a range of basic propositions with their natural meanings, a subset of which are shown in Figure 2.

3.2 A Multi-Dimensional Modal Logic

Figure 3 gives selected natural deduction rules of our logic, where Γ ranges over sets of propositions. We elide most rules for reasons of space and to focus on the novel aspects of our work, but they are standard for natural deduction presentations of dynamic logic [25]; readers less familiar with dynamic logics but familiar with classic Hoare logic [24] would find no surprises after adjusting to the different judgment form. To give some sense for those more comfortable

$$\textsc{Seq}\ \frac{\Gamma \vdash [C][C'](P)}{\Gamma \vdash [C;C'](P)} \qquad \frac{\Gamma[x \mapsto x_0], x = (t[x \mapsto x_0]) \vdash Q \quad \Gamma \vdash \mathsf{Guar}(g) \quad [\![x := t]\!] \subseteq g}{\Gamma \vdash [x := t](Q)} \qquad \text{@-T}\ \frac{}{\Gamma \vdash @_\iota(\iota)} \qquad \text{@-}\textsc{Pure}\ \frac{\vdash P}{\Gamma \vdash @_\iota(P)}$$

$$\text{@-K}\ \frac{\Gamma \vdash @_\iota(P) \quad \Gamma \vdash @_\iota(P \rightarrow Q)}{\Gamma \vdash @_\iota(Q)} \qquad \text{@-I}\ \frac{\Gamma \vdash \mathsf{Guar}(g) \quad \Gamma \vdash P \quad \Gamma \vdash \iota \quad \mathsf{Stable}(g, P)}{\Gamma \vdash @_\iota(P)} \qquad \text{@-E}\ \frac{\Gamma \vdash \iota \quad \Gamma \vdash @_\iota(P)}{\Gamma \vdash P} \qquad \textsc{Send}\ \frac{\Gamma \vdash \iota \quad \Gamma \vdash \mathsf{ActorRef}(x) \quad \Gamma \vdash y = v \quad \Gamma \vdash \mathcal{I}(t)(\iota, v)}{\Gamma \vdash [\mathsf{send}(t, x, y)](P)}$$

Figure 3. Selected rules for verifying actors

with Hoare logics, we consider the rule for sequential composition (Seq), which decomposes proofs about a sequential composition into proofs about each of the sequence commands (just as in Hoare logic). We also show one primitive rule, for assignment; this is the dynamic logic version [25] of Hoare's axiom of assignment [24], using substitution to handle where the right hand side mentions the variable being assigned, and additionally modified to ensure the action satisfies the local guarantee.

Standard for hybrid logic, a nominal is always true at the corresponding location (@-T) — here, an actor is always itself. @-Pure allows the injection of a fact that is true under no assumptions into a satisfaction modality. By itself this is not a terribly useful rule, but it works well in conjunction with the next rule. @-K is a restricted form of the typical axiom K from modal logics, which allow applying modus ponens to draw inference under a modality: intuitively if P is true at ι, and $P \rightarrow Q$ is true at ι, then Q must also be true. This is where @-Pure becomes useful: it makes it easy to inject "common-sense" implications into the satisfaction modality for a different actor, to draw further inferences from any assertions that other actor may have sent.

@-I is key: it introduces satisfaction assertions. It says that assuming P is true at the *current* actor ($\Gamma \vdash \iota$), and P is also stable with respect to the current guarantee, then it can be concluded that $@_\iota(P)$. Its dual @-E is an elimination rule for satisfaction: if P is true at ι, and ι is the current actor, then P is true at the current actor.

Finally the Send rule permits sending messages to other actors, if the invariant $\mathcal{I}(t)(\iota, v)$ for that message type can be proven of the data sent. This might include basic validity constraints (e.g., that a number to increment should be non-negative), or the requirement that the sender has witnessed some fact (like a value being a lower bound of a counter). The invariant for each message type t leaves the sender and data sent open, so it may be checked on the sender side and assumed on the recipient's side. We require that message invariants do not mention program variables outside satisfaction operators, which prevents the recipient from assuming random constraints on its local state.

Theorem 3.1 (Local Soundness)**.** *The logic is sound: For all Γ, Q, $\mathcal{M}$, w, and $\iota \in \mathsf{dom}(\mathcal{M})$, if $\Gamma \vdash Q$ and $\forall P \in \Gamma.\ \mathcal{M}, w, \iota \vDash P$, then $\mathcal{M}, w, \iota \vDash Q$.*

Proof. By induction on the derivation $\Gamma \vdash Q$. For a simple example consider Seq: there the assumptions and induction hypotheses give that $\mathcal{M}, w, \iota \vDash [C][C'](P)$, and the case requires proving $\mathcal{M}, w, \iota \vDash [C;C'](P)$. Because the semantics of the latter are given by relational composition of the semantics for C and C' this is straightforward (including additionally dealing with repetition of the guarantee on other actors' states, and the growth in their message sets). More interesting are the cases for @-K and @-I. For the former, the antecedents give that P and $P \rightarrow Q$ are true at some other actor; the assertion semantics for those assertions essentially allow repeating the reasoning for the basic modus ponens rule, but at a different actor, and additionally combining the stability information from the model interpretation of the antecedents to show stability of Q. For the latter, the antecedents □

3.3 Actor Correctness

We have yet to actually define what an actor is in our formal model. We view an actor as a set of handler routines, one for any message class (i.e., Token) the actor wishes to handle: $\mathsf{Actor} = \mathsf{Token} \rightharpoonup \mathsf{Command}$. An actor specification is a pair (ϕ, g) of an invariant over the actor's state and the actor's local guarantee. We say an actor A is correct with respect to specification (ϕ, g) under token invariants $\mathcal{I}$ — written $\mathcal{I} \vdash A : (\phi, g)$ when:

$$\frac{\forall \iota, t \in \mathsf{dom}(A).\quad \iota \wedge \phi \wedge \mathsf{Guar}(g) \wedge \mathcal{I}(t)(s, v) \wedge x = s \wedge y = v \vdash [A(t)](\phi)}{\mathcal{I} \vdash A : (\phi, g)}$$

We can extend this to correctness of a uniquely-labeled set of actors with a group specification $\mathcal{T}$ mapping actor names to actor specifications:

$$\frac{\forall (\iota, A) \in \mathcal{A}.\ \exists \phi, g.\ \mathcal{T}(\iota) = (\phi, g) \wedge \mathcal{I} \vdash A : (\phi, g)}{\mathcal{I} \vdash \mathcal{A} : \mathcal{T}}$$

This is sound with respect to interleaved handler-at-a-time semantics of a system of actors: assuming that initially every actor's invariant holds, all message invariants (from $\mathcal{I}$) hold at the sender for every message in the state, and the guarantees in $\mathcal{T}$ under-approximate the guarantees of each actor, then executing any actor's handler for any message that has been sent to it (i.e., is in the message output set of some other actor) will lead to another state where all local invariants hold, $\mathcal{T}$ underapproximates the guarantees, and all message invariants hold (including for new messages). Because our semantics is data-race free by construction, this is then equivalent to interleaving execution at a statement granularity as well.

3.4 Counters, Formally

We can model the example of Section 2 formally. Assume:

$$\begin{array}{l}\mathsf{Token} = \mathsf{LowerBound} \mid \mathsf{IncRequest}\\ \mathcal{I}(\mathsf{LowerBound})(x,y) = \exists v.\, y = v \wedge @_x(v \leq c)\\ \mathcal{I}(\mathsf{IncRequest})(x,y) = \mathsf{ActorRef}(x) \wedge 0 \leq y\end{array}$$

Then we can model the counter's one handler as:

```
c := c + y;
send(LowerBound, self, c)
```

And we can verify $\mathcal{I} \vdash \mathsf{Counter} : (0 \leq c, c \leq c')$ with the single derivation in Figure 4. Likewise, we can model the manager handler that receives updates from the counter as:

```
when (x=cntr ∧ lb < y): lb := y;
```

The manager code may be verified similarly, though we omit the derivation for space:

$$\mathcal{I} \vdash \mathsf{Manager} : \left(\begin{array}{l}(\exists v.\, lb = v \wedge @_{\mathsf{cntr}}(v \leq c)),\\ (lb \leq lb' \wedge cntr = cntr' \wedge \exists v.\, lb' = v \wedge @_{\mathsf{cntr}}(v \leq c))\end{array}\right)$$

As intended, neither actor's verification requires any relational description of any *other* actor's behavior — all knowledge of other actors comes from satisfaction operators, which witness either tautologies proven without assumptions (and therefore trivially true at all actors), or information witnessed to be stable by the actor where it is true (as in the use of @-I in the counter's proof).

4 Working with Satisfaction in Dafny

This section gives a nearly-complete encoding of the logic into Dafny, a C#-like language with integrated support for program verification, to support our claim that this is a *lightweight* extension to sequential reasoning principles. We also highlight where additional modifications would be required (such as object or reference immutability, or additional verification checks) to make the implementation sound. We assume typed asynchronous actors, as in a variant of Typed Akka [30].[1] An object of type `ActorRef<M>` is a handle to a particular actor in the system, which can be sent messages of type `M`. Figure 5 models satisfaction assertions as a higher-order predicate. We tweak the theory to allow an actor to only allow *some* of its state — which we call *publicly acknowledged* — to be referenced by other actors' assertions. This is akin to committing to a public interface for assertions about that actor — state that is *not* publicly-acknowledged can be refactored or removed without invalidating other actors' assertions. `at<T,M>(i,p)` is an assertion that the actor referred to by actor reference `i` (of type `ActorRef<M>`) exposes publicly-acknowledged state of type `T`, for which `p` is true — if `x` is the acknowledged state of the actor at `i`, then `p(x)` evaluates to true.

The next two pieces of code are two lemmas (really axioms), which in Dafny take the form of computationally-irrelevant methods. The preconditions of these "methods" are antecedents in an implication, and the postconditions are the conclusion of the lemma. `atImpl` ("at-implication") models a combination of @-Pure and @-K from the logic. The extra preconditions ensure that the conclusion (Q) is well-defined whenever the assumption (P) is well-defined. The `.requires` clauses refer to the precondition of the predicates `P` and `Q`; they are logical functions that may be applied to specific inputs (or in this case, all inputs), and so may have their own preconditions.[2] `atImpl` is axiomatized as a lemma (and later, explicitly invoked when verifying actors) because we have not taught Dafny's translation to Boogie about `at` assertions.[3] Such a modification would be desirable, but for now it also has the pedagogical benefit of highlighting where extra `at`-related reasoning is required.

Figure 5 gives the declaration for the base `Actor` class we assume, which is a simplification of the interface used by Akka. The guarantee `G()` is given as a two-state [29] predicate — a binary relation on states written in terms of an "old" and "new" state, used to constrain how state may change during execution. Akka actors have a self reference that is the actor reference for the current actor. We assume actors also carry a distinguished explicit representation of their own state, some actor-specific invariant (a single-state predicate), a method for sending messages to actors, and a method receive which handles all messages — and thus assumes and must re-establish the actor's invariant, and must ensure the updates performed adhere to `G()`.

Dafny's `twostate` invariants are useful for specifying guarantees, but when asserting a two-state invariant in a method, the old version of the state used for the check is always the state at method entry: this encoding into Dafny *does not* enforce that every atomic action obeys the guarantee, only that the aggregate effects of the receive handler do. This makes it possible in this encoding for a counter to increment the counter by 500, send that as a lower bound, then decrement by 499. The net effect of these updates is still an increment. Even asserting `G()` between every statement permits this, as even after the decrement the value is still

[1] Dafny lacks a mechanism like Java's `instanceof` to discover more precise types for an object.

[2] This was not required when defining `at` because that definition did not explicitly apply the predicate to any arguments.

[3] Experts in Dafny or dynamic frames may notice that `at` has no `reads` clause indicating which heap cells its truth relies on. This is intentional: its introduction is restricted to stable predicates, whose stability is enforced elsewhere, and need not be checked explicitly when manipulating `at` assertions.

$$\dfrac{\dfrac{\ldots \vdash \mathsf{self} \qquad \ldots \vdash \mathsf{ActorRef}(x) \qquad \dfrac{\ldots \qquad \dfrac{\ldots}{\ldots \vdash @_{\mathsf{self}}(c \leq c)}\ @\text{-I}}{\ldots \vdash \mathsf{ActorRef}(x) \land \mathsf{Number}(c) \land @_{\mathsf{self}}(c \leq c)}\ \land\text{-I}}{\mathsf{self} \land 0 \leq c_0 \land \mathsf{Guar}(c \leq c') \land \mathsf{ActorRef}(x) \land 0 \leq y \land c = c_0 + y \vdash [\mathsf{send}(\mathsf{LowerBound}, x, c)](\phi)}\ \textsc{Send}}{\dfrac{\mathsf{self} \land 0 \leq c \land \mathsf{Guar}(c \leq c') \land \mathsf{ActorRef}(x) \land 0 \leq y \vdash [c := c + y][\mathsf{send}(\mathsf{LowerBound}, x, c)](\phi)}{\mathsf{self} \land 0 \leq c \land \mathsf{Guar}(c \leq c') \land \mathsf{ActorRef}(x) \land 0 \leq y \vdash [c := c + y; \mathsf{send}(\mathsf{LowerBound}, x, c)](\phi)}\ \textsc{Seq}}\ \textsc{Assign}$$

Figure 4. Proving correctness of the counter actor

larger than it was initially. This is one place Dafny (or a similar system) would require change to soundly implement our calculus (essentially, the guarantee check from the axiom rule is not performed here). However, since Dafny already includes two-state predicates, the change would be to enforce the existing checks between more pairs of states, not building new functionality. This would affect the programming model, but is within the reach of current verification systems.

Readers may have noticed that the method signature for sending messages is not as restrictive as that in the Send rule in Section 3. Instead of directly encoding the invariant map $\mathcal{I}$ from the formal calculus, we provide a `Witness` class to bundle data and assertions when sending messages. The implementations can choose the invariant over their data, which necessarily includes an actor reference to the sender and thanks to the `reads this` clause is required to only mention message state. This is how we impose stability proofs and control introduction of satisfaction assumptions (i.e., @-I) as well. `Witness`es have a method `stability` that must prove stability of the message's predicate with respect to the given actor's guarantee. The witness constructor accepts a direct *object* reference to the actor that will send the message, from which all state is accessible. Implementations of `Witness` will choose more selective preconditions depending on the predicate, and must prove the predicate holds at that actor. Typically the only way to do this is for an actor to establish some fact about its state locally, and pass itself (`this`) into the witness constructor. The constructor can then use the `introAt` (@-I) axiom to convert this information into a satisfaction assertion. Recipients, instead of having complex preconditions on receive, can simply use the postcondition of the static `Unpack` method.

This touches on the other class of extensions Dafny would require to soundly implement our calculus, which is something akin to the reference immutability type systems already present in some actor systems [8, 21]. Two things could go awry with this `Witness` construction in Dafny today: the constructor could use `Unpack` itself to prove the satisfaction assertion without sufficient evidence, or an actor could send the *object* reference to itself to another actor, enabling data races (and specifically, letting another actor observe possibly-unstable properties). Reference immutability could fix both issues, as well as enforcing message immutability: the signatures for sending or unpacking `Witness`es could be refined to require deeply-immutable inputs. This prevents actors from sending their own `this` reference, since that reference must be mutable in the receive handler. It also prevents a `Witness` constructor from using `Unpack`, since the receiver would be mutable inside the constructor. Actor code would then need to freeze the witness *after* construction, send it as immutable, and the recipient could then use `Unpack`.[4]

4.1 Counter and Manager in Dafny

If this sounds very abstract, seeing code for the counter and manager example may help. Both are implemented as module refinements of the `DafnyActor` module. The counter also refines the witness, whose constructor accepts a (counter) actor and copies out the current value as a new lower bound, establishing the witness predicate. The stability lemma includes a workaround to state that the lower bound of the witness does not change; Dafny lacks a way to specify that all fields of an object remain the same after construction, but an extension with reference immutability could assume this. The counter actor itself has the expected behavior: its message handler replies to the sender with a new witness for the lower bound. Both actors have the invariants and guarantees from Section 3.4. The manager unpacks the witness to be able to assume the message contains a lower bound, and uses `atImpl` to guide Dafny's unmodified core to transfer this to a new lower bound. A slightly elaborated version of this code is available online.[5]

4.2 Generality

The approach taken here could be adapted to any sequential verification system capable of encoding (or being extended to encode) guarantee relations, stability checks, higher-order predicates, immutability, and checks that every individual step satisfies the guarantee. Liquid Haskell [46] and KeY [1] can encode all but the last natively; we considered them for our axiomatization experiment, but Haskell has no well-established actor framework to mimic (with an eye towards

[4]This ignores the opportunity to send externally unique object graphs safely between actors, but this could also be accommodated: mutable references in these systems cannot be considered externally unique, either.

[5]https://gist.github.com/csgordon/b9173c2b28099e8353c36eb19c058691

```
class {:extern} ActorRef<Ms> {}
predicate at<T,Ms>(i:ActorRef<Ms>, p:(T ⇝ bool))
lemma atImpl<T,M>(c:ActorRef<M>,
                  P:T⇝bool, Q:T⇝bool)
 requires ∀ x:T • P.requires(x) ⟹ Q.requires(x)
 requires ∀ x:T • P.requires(x) ⟹ P(x) ⟹ Q(x)
 ensures at(c, P) ⟹ at(c, Q)
/*Utility class for packing a sender & message*/
class MsgBox<T,U> {
  var sender: ActorRef<T>
  var msg: U
  constructor(s:ActorRef<T>, m:U) {
    sender := s;
    msg := m;
} }
abstract module DafnyActor {
  type State
  type Msgs
  twostate predicate stable(a:Actor,P:State⇝bool)
    reads a, P.reads {
      (old(P.requires(a.state)∧P(a.state))∧a.G())
      ⟹ (P.requires(a.state))∧P(a.state)
  }
  class Actor {
    twostate predicate G() reads this
    function method self():ActorRef<Msgs>
      reads this
    var state: State
    constructor {:extern} () ensures inv()
    predicate inv() reads this
    method send<T>(dest: ActorRef<T>, msg:T)
    method receive(message: Msgs)
      modifies this
      requires inv()
      ensures inv()
      ensures G()
  }
  class Witness {
    var loc: ActorRef<Msgs>
    predicate P(s:State) reads this
    twostate lemma stability(x:Actor)
      ensures stable(x, P)
    constructor(r:Actor)
      ensures P(r.state) ∧ at(r.self(), P)
    lemma introAt(r:Actor)
      requires r.self() = loc ∧ P(r.state)
      ensures at(r.self(), P)
    { assume at(r.self(), P); }
    static method Unpack(w:Witness)
      ensures at(w.loc, w.P)
    { assume at(w.loc, w.P); }
} }
```

Figure 5. Core Dafny definitions of satisfaction modality, idiomatic combination of @-Pure with @-K, and actor classes.

```
module CounterMod refines DafnyActor {
  type State = nat
  type Msgs = MsgBox<Witness,nat>
  class Witness {
    var lb: nat
    predicate P(s:State) { lb ≤ s }
    twostate lemma stability(x:Actor)
      ensures stable(x, P) {
      assume lb = old(lb);//Immutability workaround
    }
    constructor(r:Actor)
      ensures P(r.state) ∧ at(r.self(), P) {
      lb := r.state;
      loc := r.self();
      new;
      introAt(r);
  } }
  class Actor {
    constructor() ensures inv() { state := 0; }
    twostate predicate G() reads this {
      old(this.state) ≤ state
    }
    predicate inv() { 0 ≤ this.state }
    method receive(message: Msgs) {
      state := state + message.msg;
      var w := new Witness(this);
      send(message.sender, w);
} } }
module ManagerMod refines DafnyActor {
  type State = nat
  type Msgs = CounterMod.Witness
  class Actor {
    var child:
      ActorRef<MsgBox<CounterMod.Witness,nat> >
    twostate predicate G() reads this {
      old(child)=child ∧ old(state) ≤ state ∧
      at(child, (s:nat) reads this ⇒ state ≤ s)
    }
    predicate inv() {
      at(child, (s:nat) reads this ⇒ state ≤ s)
    }
    method receive(message: Msgs)
    {
      var sender := message.loc;
      CounterMod.Witness.Unpack(message);
      atImpl(sender, message.P,
        (s:nat) reads message ⇒ message.lb ≤ s);
      if (sender = child ∧ state < message.lb) {
        state := message.lb;
        atImpl(child,
          (s:nat) reads message ⇒ message.lb ≤ s,
          (s:nat) reads this ⇒ this.state ≤ s);
      }
} } }
```

Figure 6. Dafny code for Counter and Manager

extraction of verified actors), and KeY's specification language has great power at the cost of great verbosity. There are known extensions to KeY's foundations that support the required guarantee checks as well [4], but they are not implemented in KeY. Alternatively, construction of a verification tool for a language like Pony that already has reference immutability [8] would require only small extensions beyond standard sequential verification tools.

5 Related Work

Some related work was addressed earlier in the course of presenting background material for our technical development. This section focuses on two further clusters of related work: means of proving correctness for actor programs, and related variants of dynamic and/or hybrid logic.

Actor Correctness There are many possible approaches to verifying actor programs. The most successful work in this space has been the use of reference capabilities to prevent data races [2, 8, 21] or ordering races [3] in actor systems, but these are only capable of controlling interference between actors, not of proving any sort of invariants.

Work on static analysis on actor programs [9, 14, 41] generally requires analysis of a closed program — one where all actors appear — rather than analyzing open programs consisting of some actors but not all (as required for separately verifying libraries). Recently Desai et al. [10] addressed the open program issue by modeling actors and an environment abstraction as input/output automata [31], then performing automata-theoretic refinement checking against an abstract specification. We are unaware of work applying program logics specifically to actor programs.

Most work on rely-guarantee reasoning separates rely and guarantee relations (even in work that uses transition systems rather than binary relations [27, 35, 42]). In general this makes sense, as different threads may have asymmetric roles (e.g., producer and consumer threads) in a shared-memory setting. The cost of this is that every thread's proof must reason explicitly at times about *other* threads' behavior. We made a specific choice to encapsulate all relational specification of an actor (its guarantee relation) to the actor itself. In principle one could imagine providing separate rely and guarantee relations, where different handles to each actor granted the holder the rights to send different messages affecting the recipient's state differently, in a manner similar to rely-guarantee references [19, 20] or rely-guarantee protocols [33, 34]. This might grant additional verification power, but at the cost of substantial complexity: rely relations for even modest data structures (e.g., the union-find data structure studied by Gordon et al. [20]) can be quite complex, and such an approach would require *clients* of an actor to to check that their local assertions were stable with respect to the rely relations specific to their handle to a peer actor. This not only complicates the amount of reasoning actors must do about each other, but hurts modularity as well: if an actor makes some new action possible, any actor making assumptions about it must have its stability proofs redone even if all assertions it makes are stable.

Modal Logics As mentioned earlier, dynamic logic [38] is a well-established form of weakest precondition approach to imperative program correctness, underlying Hoare Logic in a way that has fed into recent developments [1, 27]. Hybrid logic is a smaller, but also well-established [6, 7] class of modal logics. The logic we presented in Section 3 is a combination of these two forms of logic, technically classified as a multi-dimensional modal logic [32], since the points at which formulas are evaluated have internal structure with different pieces addressed by different modalities in the logic. Our model diverges from common practice in multi-dimensional modal logics: while most n-dimensional modal logics take points of formula evaluation to be n-tuples of a common world state, our (2-dimensional) points of evaluation are heterogeneous: a collection of actors, and a choice of a particular actor's point of view.

There are many modal logics combining state change and notions of place (e.g., spatio-temporal logics [5] or dynamic epistemic logic [45]), but to the best of our knowledge we are the first to propose using hybrid logic for places in dynamic systems, or to address assertion stability in a dynamic logic. Dynamic epistemic logic is probably most similar to our work, as a combination of dynamic logic with *epistemic logic* (logic of what participants have what knowledge). However, this branch of logic typically focuses on reasoning about what knowledge is preserved across specific actions that modify the world rather than limiting the logic to stable knowledge based on other restrictions on allowable actions. It typically also concerns global knowledge rather than knowledge of facts *about* individuals, permitting inferences such as "if a knows P, then I know P," the equivalent of which in our system would be "if P is true at a, then P is true here" which is obviously incorrect if P's truth depends on which actor considers the formula.

6 Conclusions & Future Work

This paper outlines the core of an approach to enable deductive verification of actor systems with only modest extensions beyond established techniques, and demonstrates some promise, but the version presented here is limited. We have not considered actor creation or use of "become" to switch behaviors; both should be possible as long as the new behaviors satisfy the relevant guarantee. We have also not given proofs about use of local actor state with heaps; the Dafny prototype should be sound if extended so the only state that could be mentioned in `at` assertions was immutable (e.g., immutable `Witnesses`), but further work is needed to both prove this and determine if additional flexibility is possible. We have also not considered failure and restarting of actors, which is likely to require some extension. Moreover, further investigation of the technique's practical limits are needed to determine how broadly useful this approach is.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1844964 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1844964). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. 2016. *Deductive Software Verification–The KeY Book*. Springer.

[2] Mehdi Bagherzadeh and Hridesh Rajan. 2015. Panini: A Concurrent Programming Model for Solving Pervasive and Oblivious Interference. In *MODULARITY 2015*.

[3] Mehdi Bagherzadeh and Hridesh Rajan. 2017. Order Types: Static Reasoning About Message Races in Asynchronous Message Passing Concurrency. In *AGERE*.

[4] Bernhard Beckert and Daniel Bruns. 2013. Dynamic logic with trace semantics. In *International Conference on Automated Deduction*.

[5] Brandon Bennett, Anthony G Cohn, Frank Wolter, and Michael Zakharyaschev. 2002. Multi-dimensional modal logic as a framework for spatio-temporal reasoning. *Applied Intelligence* 17, 3 (2002), 239–251.

[6] Patrick Blackburn and Jerry Seligman. 1995. Hybrid languages. *Journal of Logic, Language and Information* 4, 3 (1995), 251–272.

[7] Torben Braüner. 2010. *Hybrid logic and its proof-theory*. Springer.

[8] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *AGERE*.

[9] Jean-Louis Colaço, Mark Pantel, and Patrick Sallé. 1997. A set-constraint-based analysis of actors. In *Formal Methods for Open Object-based Distributed Systems*. Springer, 107–122.

[10] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 159.

[11] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457.

[12] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP*.

[13] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *ESOP*.

[14] Emanuele D'Osualdo, Jonathan Kochems, and C-H Luke Ong. 2013. Automatic verification of Erlang-style concurrency. In *SAS*.

[15] Xinyu Feng. 2009. Local Rely-Guarantee Reasoning. In *POPL*.

[16] Michael J Fischer and Richard E Ladner. 1979. Propositional dynamic logic of regular programs. *Journal of computer and system sciences* 18, 2 (1979), 194–211.

[17] George Gargov and Valentin Goranko. 1993. Modal logic with names. *Journal of Philosophical Logic* 22, 6 (1993), 607–636.

[18] Valentin Goranko. 1996. Hierarchies of modal and temporal logics with reference pointers. *Journal of Logic, Language and Information* 5, 1 (1996), 1–24.

[19] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. 2013. Rely-Guarantee References for Refinement Types Over Aliased Mutable Data. In *PLDI*.

[20] Colin S. Gordon, Michael D. Ernst, Dan Grossman, and Matthew J. Parkinson. 2017. Verifying Invariants of Lock-free Data Structures with Rely-Guarantee and Refinement Types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 3 (July 2017).

[21] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*.

[22] David Harel. 1979. First-order dynamic logic.

[23] Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger. 1973. Actor Induction and Meta-Evaluation. In *POPL*.

[24] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[25] Furio Honsell and Marino Miculan. 1995. A natural deduction approach to dynamic logic. In *International Workshop on Types for Proofs and Programs*. Springer, 165–182.

[26] C. B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 4 (Oct. 1983), 596–619.

[27] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

[28] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 348–370.

[29] K. Rustan M. Leino and Wolfram Schulte. 2007. Using History Invariants to Verify Observers. In *ESOP*.

[30] Inc. Lightbend. 2019. Akka Actors. https://akka.io

[31] Nancy A. Lynch and Mark R. Tuttle. 1987. Hierarchical Correctness Proofs for Distributed Algorithms. In *PODC*.

[32] Maarten Marx and Yde Venema. 1997. *Multi-dimensional modal logic*. Vol. 4. Springer Science & Business Media.

[33] Filipe Militão, Jonathan Aldrich, and Luís Caires. 2014. Rely-Guarantee Protocols. In *ECOOP*.

[34] Filipe Militão, Jonathan Aldrich, and Luís Caires. 2016. Composing Interfering Abstract Protocols. In *ECOOP*.

[35] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and GermÃąn AndrÃľs Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*.

[36] Susan Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* (1976), 319–340. Issue 6.

[37] Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS*. IEEE.

[38] Vaughan R Pratt. 1976. Semantical consideration on Floyd-Hoare logic. In *FOCS*.

[39] Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *ESOP*.

[40] Krister Segerberg. 1973. Two-dimensional modal logic. *Journal of Philosophical logic* 2, 1 (1973), 77–96.

[41] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. 2017. Mailbox Abstractions for Static Analysis of Actor Programs. In *ECOOP*.

[42] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *ICFP*.

[43] Viktor Vafeiadis. 2007. *Modular Fine-Grained Concurrency Verification*. PhD Thesis. University of Cambridge.

[44] Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *Concurrency Theory (CONCUR)*.

[45] Hans Van Ditmarsch, Wiebe van Der Hoek, and Barteld Kooi. 2007. *Dynamic epistemic logic*. Vol. 337. Springer Science & Business Media.

[46] Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded Refinement Types. In *ICFP*.