Global Specifications for Distributed Protocols

Abstract. Distributed protocols are an essential part of modern computing infrastructure, yet difficult to implement correctly. While there has been much progress in verifying the design of protocols, end-to-end verification of real-world protocol implementations remains a challenging endeavour. To address this problem, we propose a new global choreographic communication language for specifying distributed protocols which is amenable to end-to-end verification. A novel aspect of this language lies in its expressive power while keeping the protocol specification concise via its support for variable assignment, quantification over participant sets, disjunctive choices and conditionals. Another novelty of our work is the integration of such global specifications to TLA$^+$ to be verified, and the automatic decomposition into atomic actions used for instrumenting and monitoring an implementation for safety and refinement violations. This forms the basis of a lightweight runtime verification method that we show to be readily applicable to existing implementations incurring only a modest performance overhead and low instrumentation effort.

1 Introduction

Distributed systems are an essential part of modern computing infrastructure. At their heart are *protocols* which express how multiple nodes should interact to achieve a shared goal. It is critical for these protocols to have corresponding *safe and correct implementations*, as they make up core components of larger systems. However, this is notoriously difficult to ensure in the presence of faults, asynchrony, and the combinatorially-large state spaces that result.

The verification of distributed protocols themselves has been well-studied [46,20,34,52,50]. A common approach is to specify a protocol as some form of transition system, characterize its state space using inductive invariants, and then formulate correctness as safety and liveness properties. A notable mention in this line of works is TLA$^+$ [28], a lightweight formal method successfully applied in industry [38].

In contrast, end-to-end verification remains a challenging endeavor. With few exceptions [21,52], the vast majority of distributed systems today are unverified. Modern approaches rely on deductive verification with specialized logics [50,25] and stepwise refinement [21] or systematic testing [17]. The high cost of these approaches though – on the order of person-years for realistic developments [21,53] – generally make them worthwhile only for critical systems. We think that end-to-end verification should be widely applicable, if in a vastly simplified form [48] such as *rigorous testing*. Since implementations evolve frequently and in unpredictable ways, it is important to come up with ways to keep a protocol and its implementation in sync, or else the latter might diverge and be eventually abandoned [1]. Compared to abstract protocol descriptions, implementations

are written in featureful programming languages with dynamic and higher-order features. To solve this problem, we must bridge the gap between them and the mathematical models that underlie specifications.

We thus design a simple method for checking a specification against an implementation, to ensure firstly that the latter preserves safety properties, and secondly that it continues to follow the specification as it evolves. The approach supports incremental adoption, to ensure it is usable in mature but unverified systems, which have typically received significant engineering investment, and for which a clean-room rewrite or extensive restructuring would a non-starter.

To achieve this, we design a high-level specification language for distributed protocols. This language is *global* and communication-centric (à la multiparty session types [22,13]), capturing the communication structure of a protocol from a neutral perspective and allowing specifications to closely resemble informal natural-language descriptions. A global protocol involving multiple interacting parties can then be projected to individual party-local specifications and translated into a set of atomic actions. From here, it is compiled into TLA$^+$ for verification using either model checking [55,27] or theorem proving [12], and used to generate a safety monitor for runtime verification of an existing implementation.

This gives us a lightweight way of testing conformance of an implementation to a protocol, without requiring a rewrite or invasive changes to an implementation. We see monitoring an implementation as an incomplete but principled and practical testing method, a middle ground between testing and verification.

Contributions. The contributions of our work are as follows:

1. A multiparty, communication-centric specification language for distributed protocols which allows natural and succinct specifications. It differs from existing languages in its support for quantification over set of parties, state, conditionals, and structured concurrency (Sec. 3).
2. Sound mechanisms for projecting global specifications into local ones and decomposing them into high-level, local atomic actions, with control flow and other constraints encoded as preconditions. This treatment supports seamless integration with TLA$^+$ for checking protocol safety, thus offering users the option to define additional properties to be checked. (Sec. 4).
3. A runtime verification method which supports incremental integration into practical programs, allowing them to be monitored for both protocol conformance as well as adherence to safety properties (Sec. 5).

2 Overview

Fig. 1 offers an overview of our approach. A global protocol specification is first projected onto local specifications, each of which is decomposed into a set of atomic actions. Given user-provided safety properties, actions may serve as TLA$^+$ specifications for the verification of the protocol, or as an executable monitor to validate an implementation. We highlight parts of the workflow by specifying the classic two-phase commit protocol [31], in which a set of database replicas collectively agree on whether or not to commit a distributed transaction.

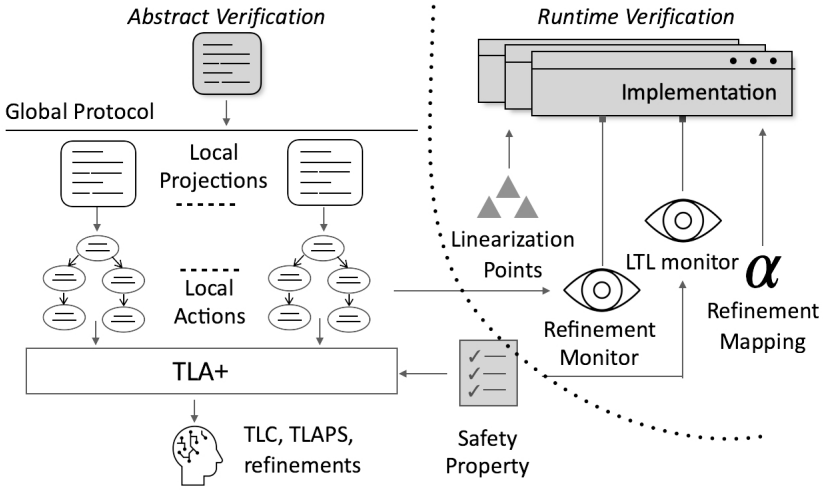


Fig. 1: Overview: the protocol is checked with TLA$^+$; implementation conformance is checked at runtime, assuming the user-provided information in gray.

```
1  forall c in C
2    (forall p in P
3       c→p : prepare;
4       (p→c : prepared
5        ∨
6        p→c : abort;
7        c.has_aborted = true));
8    (!(c.has_aborted) ⇒
9       (forall p in P
10         c→p : commit;
11         p→c : commit_ack(p=p);
12         committed =
13           union(committed, {p}))
14     ∨
15     c.has_aborted ⇒
16       (forall p in P
17          c→p : abort;
18          p→c : abort_ack(p=p);
19          aborted =
20            union(aborted, {p})))
```

Fig. 2: Two-phase commit protocol

Specifying Protocols. The protocol is initiated by the *coordinator*, a distinguished node which first sends requests to a set of *participant* nodes, asking them to *prepare* to commit a transaction. The participants must then decide independently whether or not to commit; a node may decide not to, for example, because it has detected a conflict. Each participant replies with its decision to the coordinator, which then enters the second phase of the protocol, deciding to commit the transaction if and only if *all* participants decided to commit. Whatever its decision, it performs another broadcast to inform the participants, which end the protocol by acknowledging the decision.

The protocol is shown in Fig. 2. We assume the existence of two *roles* `P` and `C`: finite sets of indistinguishable nodes which posses a unique pattern of interactions in a protocol.

The atomic actions of a global protocol are transmissions, identified by $\rightarrow$, which separates sender from receiver, followed by the message description, and assignments which modify node-local state. `forall` allows quantification over

roles, so line 3 is really a broadcast from all nodes `c` in the (singleton) coordinator set `C` to all participant nodes `p` in `P`. There is no obligation that this be done sequentially, so every send in the broadcast may happen in parallel.

Sequential composition (denoted by ;) expresses program order when statements concern the same nodes, e.g. lines 6-7, where the coordinator performs a state change after receiving an abort message from a participant.

```
21 (forall p in P
22    →p : prepare;
23    (p→ : prepared
24     ∨
25     p→ : abort;
26     has_aborted = true));
27 (!(has_aborted) ⇒
28    (forall p in P
29       →p : commit;
30       p→ : commit_ack(p);
31       committed =
32         union(committed, {p})))
33  ∨
34  has_aborted ⇒
35    (forall p in P
36       →p : abort;
37       p→ : abort_ack(p);
38       aborted =
39         union(aborted, {p})))
```

(a) Coordinator projection

```
40 forall c in C
41   c→ : prepare;
42   (→c : prepared
43    ∨
44    →c : abort);
45   (c→ : commit;
46    →c : commit_ack(p=self)
47    ∨
48    c→ : abort;
49    →c : abort_ack(p=self))
```

(b) Participant projection

Fig. 3: Local Projections

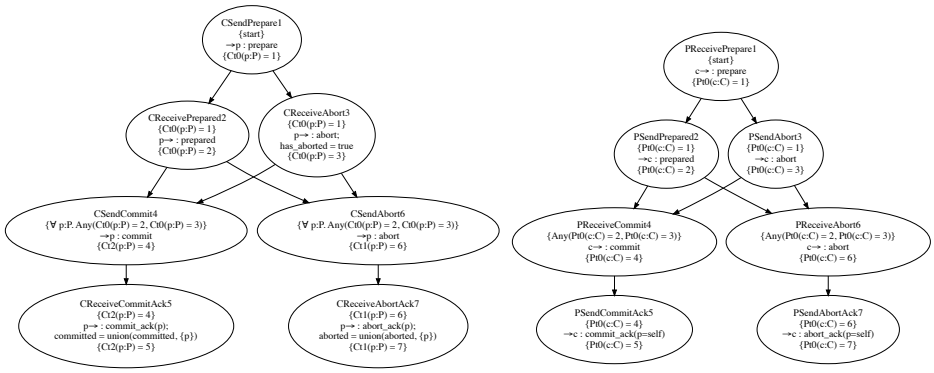
Common concurrency patterns may be succinctly expressed using ; and **forall**. The ; operator at line 7 expresses a *join* over multiple conceptual threads of execution involving the nodes in `p`. This captures the blocking nature of the protocol and divides its two phases.

Finally, ∨ denotes disjunctive composition of subprotocols. Decisions are expressed using ⟹, which denotes a subprotocol that only executes if the guard is true at that instant and otherwise does nothing. The disjunction at line 14 is guarded with disjoint conditions, and so is deterministic. The one at line 5, on the other hand, is nondeterministic; a participant may choose to commit or not.

The specification language is typed. Party types are indexed by role and thus distinguished from one another. Sets are assumed to contain elements of a single type. From this we can infer, for example, that `committed` is a set of nodes of role `P`.

We also infer *locations* for all expressions, reminiscent of the *tier-splitting* that tierless programming languages do [11]. Our language is designed for modeling distributed systems and communication, so there cannot be shared memory or mutable global variables, and every occurrence of mutable state must be node-local. Our type system rules out specifications with shared state and allows us to deduce that `committed` and `aborted` *must* be local to coordinator nodes.

Distributed protocols must often uphold a safety property: a predicate on the protocol states which should hold in all executions starting from the initial state. The safety property for two-phase commit is that every node either commits or aborts; it is impossible that we observe one node that has committed and another



(a) Coordinator actions

(b) Participant actions

Fig. 4: Actions

that has aborted. Using node-local protocol state, we may express it as:

$$committed \cup aborted = P \implies committed = \emptyset \vee aborted = \emptyset$$

We next demonstrate how the protocol and its safety property can be used to check an implementation for conformance. By nature, implementations must deal with low-level communication, and are consequently written from the perspective of a single party, obscuring the overall picture of what node interactions look like. This key shortcoming is what multiparty specification languages address.

To define a correspondence between specification and implementation, we project the specification on each party. The coordinator and participant projections are shown in Fig. 3b and Fig. 3a, respectively.

In the coordinator projection, the quantification over `C` has vanished; this is because the protocol is now expressed from the perspective of a node in `C`. Transmissions, previously binary operations between two nodes, are now unary, with the implicit operand taken to be the current node. Likewise, in the participant projection, the deterministic choice at line 27 is not present anymore; this is because it depends on the coordinator-local state, hence only the coordinator may initiate it. In other words, it is a choice that is internal to the coordinator.

Next, we decompose each local projection into a set of atomic *actions*—an intermediate representation of a protocol. These are broadly similar to the eponymous actions of TLA, intuitively corresponding to transitions of the protocol's state machine. The set of actions may be visualized as a graph, as shown in Fig. 4a and Fig. 4b, where edges denote program order. Like the projections, they mirror each other.

Each action is decorated with a precondition which encodes the control flow of the protocol projection it came from. For example, the precondition of the action *CSendCommit4* in Fig. 4a, which corresponds to the state change the coordinator effects when it sends a `commit` message to a participant, captures the requirement that all `P` threads of communication must be completed by then (the *join* we mentioned earlier, at line 26 of Fig. 3a).

Actions correspond intuitively to atomic steps an implementation takes. Along with the safety property of the protocol, they are used to check an implementation for conformance, while translating them to TLA$^+$ facilitates model checking, protocol verification, and refinement. Coupled with user annotations of commit points, they enable monitor checking for refinement and safety violations in an actual implementation. Below are the actions for two-phase commit:

$$
\begin{aligned}
\mathit{Next} \triangleq \lor\ & \exists\ \mathit{self} \in C : \exists\ p \in P : \\
& (\mathit{CSendPrepare1}(\mathit{self}, p) \lor \mathit{CReceivePrepared2}(\mathit{self}, p) \lor ...) \\
\lor\ & \exists\ \mathit{self} \in P : \exists\ c \in C : \\
& (\mathit{PReceiveCommit5}(\mathit{self}, c) \lor \mathit{PReceiveAbort6}(\mathit{self}, c) \lor ...)
\end{aligned}
$$

This not only ensures that the implementation obeys safety properties specified by the protocol, but also that it corresponds closely to the specification, allowing them to evolve in tandem. This way, specifications written in our language serve as a bridge between the high-level communication-centric protocol and a low-level implementation.

3 Global Specifications

In this section we formally introduce the global choreographic communication language highlighting its novel features and describing its semantics. Additionally, we also introduce a type system to statically validate the global protocol for subtle design bugs.

3.1 Syntax and Semantics

The syntax of our specification language is given in Fig. 5. It is designed to express role-centric distributed protocols, where multiple classes of parties communicate and synchronize to achieve a shared goal. e is the type of pure expressions, which consist of qualified variables, values, sets, and function application.

The simplest protocols G are assignments, which change participant-local state, and transmission of a message of type m, allowing participants to effect and respond to state changes in other participants. To simplify the presentation, we assume without loss of generality that functions have exactly two arguments (akin to binary operators), and messages only one field. We also explicitly qualify assignments with the party they are local to for clarity.[1]

More complex protocols are made of simple ones composed sequentially, in parallel, disjunctively (allowing the uniform expression of both nondeterminism and internal/external choice), with the blocking guard $\Rightarrow^*$ (read as *when*), and

[1] In our implementation, roles and qualifiers may in many cases be inferred. While it is interesting to think of state as being intrinsic to the protocol itself, and the inference process as a way to partition it for the distributed, message-passing setting, this feature is not essential to our theory.

$$
\begin{array}{rll}
G ::= & \mathit{skip} & \text{No-op} \\
\mid & q.v = e & \text{Assignment} \\
\mid & s \to r : m(v = \nu) & \text{Transmission} \\
\mid & G \,||\, G & \text{Parallel} \\
\mid & G \,;\, G & \text{Sequencing} \\
\mid & G \vee G & \text{Disjunction} \\
\mid & e \Rightarrow^* G & \text{When} \\
\mid & \forall v \in R.\, G & \text{Quantifiers}
\end{array}
\qquad
\begin{array}{rll}
\nu ::= & i \mid b & \text{Int, Bool} \\
e ::= & q.v \mid \nu & \text{Variables, values} \\
\mid & \{e\} \mid f(e_1, e_2) & \text{Sets, functions} \\
\tau ::= & \mathit{Int} \mid \mathit{Bool} & \text{Values} \\
\mid & \mathit{set}\ \tau & \text{Sets} \\
\mid & (\tau_1, \tau_2) \to \tau_3 & \text{Functions} \\
\mid & \mathit{party}\ R & \text{Parties} \\
\ell ::= & \mathit{global} \mid \mathit{party}\ R & \text{Locations}
\end{array}
$$

$$R \in \mathcal{R} \qquad p, q, r, s \in R$$

Fig. 5: Global protocol syntax and types

using quantification over parties. We sometimes make use of the *nonblocking* guard $e \Rightarrow G$ (read as *if*), which stands for $e \Rightarrow^* G \vee \neg e \Rightarrow^* \mathit{skip}$.

We assume an asynchronous communication model with nonblocking sends and blocking receives. The network is assumed to be unreliable, and messages may be dropped, duplicated, reordered, or arbitrarily delayed. This is to allow a wide variety of practical protocols to be modeled. We may recover full synchrony by blocking, e.g. $e \Rightarrow^* G$, or partial synchrony when $e = \mathit{true}$, to represent an arbitrary timeout.

We organize participants in a distributed protocol into a set $\mathcal{R}$ of *roles*. A role $R \in \mathcal{R}$ is defined by a unique pattern of interaction, or behavior; operationally, we take a role to be a set of *parties* $p \in R$ with that behavior.

Following established traditions, the semantics of a distributed protocol is a state-transition system $(\Sigma, \longrightarrow)$. The state of the entire system Σ consists of the local states of parties $\sigma = \{v_1 = e_1, ...\}$, as well as the *message soup*, which models the state of the network in the form of a multiset of in-flight messages $\{\!\{m(v = \nu), ...\}\!\}$. $\langle \Sigma, G \rangle \longrightarrow \langle \Sigma', G' \rangle$ is a big-step reduction relation, which we define shortly.

We first introduce a relation $v \downarrow \ell$ (read as *located*) which labels each variable v as residing on either parties of a given role R, or all parties, in which case v is immutable and readable in a *global* capacity. $\downarrow$ extends to expressions straightforwardly, holding if *all* variables in a given expression reside in the same location. Consequently, well-formed expressions have exactly one location; we discuss more about this in the following sections, where we present a type system which enforces this property and other well-formedness conditions.

We also define *structural congruence* $\equiv$ as the smallest congruence relation on global protocols including the following equations.

$$\mathit{skip} \,||\, G \equiv G \qquad G_1 \,||\, G_2 \equiv G_2 \,||\, G_1 \qquad (G_1 \,||\, G_2) \,||\, G_3 \equiv G_1 \,||\, (G_2 \,||\, G_3)$$
$$G_1 \vee G_2 \equiv G_2 \vee G_1 \qquad (G_1 \vee G_2) \vee G_3 \equiv G_1 \vee (G_2 \vee G_3)$$
$$\mathit{skip} \,;\, G \equiv G \qquad G \,;\, \mathit{skip} \equiv G \qquad (G_1 \,;\, G_2) \,;\, G_3 \equiv G_1 \,;\, (G_2 \,;\, G_3)$$
$$\forall p \in P.\ \mathit{skip} \equiv \mathit{skip} \qquad e \Rightarrow^* \mathit{skip} \equiv \mathit{skip}$$

$$\textsc{G-FORALL}\quad \frac{p = \{p_1, p_2, ...\}}{\langle \Sigma, \forall v \in p.\ G\rangle \longrightarrow \langle \Sigma, G[p_1/v] \mid\mid G[p_2/v] \mid\mid ...\rangle}$$

$$\textsc{G-ASSIGN}\quad \frac{v, e \downarrow \mathit{party}\ R \qquad q \in R \qquad \Sigma(q) = \sigma \qquad \langle \sigma, e\rangle \Downarrow \nu \qquad \sigma' = \sigma[\nu/v]}{\langle \Sigma, q.v = e\rangle \longrightarrow \langle \Sigma[\sigma'/q], \mathit{skip}\rangle}$$

Fig. 6: Operational semantics (select rules)

We now define $\longrightarrow$ in Fig. 6. $\langle \sigma, e\rangle \Downarrow \nu$ is a standard call-by-value evaluation relation for pure expressions. **`forall`** is the only binder for party variables, and the rule for it shows that quantification is conceptually equivalent to multiple parallel instances of the protocol, one for each party being quantified over. Assignments modify the state of q, where v is located. Transmission is asynchronous, and so reduces in two steps, the first moving a message into the message soup, and the second consuming it, then binding the variable corresponding to its field in the recipient. Sequential composition reduces subprotocols from left to right, while parallel composition may nondeterministically reduce either side. With disjunction, exactly one reducible alternative is taken. *when* depends on $\downarrow$ and only has a defined meaning on parties where the conditional expression is located.

$$\textsc{T-ASSIGN}\quad \frac{\begin{array}{c}\Gamma \vdash_\downarrow v : \langle \tau, \ell_v\rangle \\ \Gamma \vdash_\downarrow e : \langle \tau, \ell_e\rangle \\ \Gamma \vdash_\downarrow q : \langle \mathit{party}\ R, \ell_q\rangle \\ \ell_v \sim \ell_e \sim \ell_q \sim \mathit{party}\ R\end{array}}{\Gamma \vdash q.v = e}$$

$$\textsc{T-TRANSMISSION}\quad \frac{\begin{array}{c}\Gamma \vdash_\downarrow s : \langle \mathit{party}\ R_s, \ell_s\rangle \qquad \Gamma \vdash_\downarrow r : \langle \mathit{party}\ R_r, \ell_r\rangle \\ \Gamma \vdash_\downarrow e : \langle \tau, \ell\rangle \qquad \mathit{party}\ R_s \sim \ell_s \sim \ell_r \sim \ell\end{array}}{\Gamma \vdash s \to r : m(v = \nu)}$$

Fig. 7: Type system (selected rules)

3.2 Type system

A type system for a specification language may seem extraneous: session-typed systems typically only have very simple kind systems, if at all. However, the addition of state introduces new failure modes, such as the use of information which may not (yet) be available to a given party. Such problems are not obvious in a multiparty language but can be prevented with rigorous reasoning.

Furthermore, nontrivial protocol specifications are thousands of lines long and may themselves hide bugs [45]. It is therefore useful to validate them statically, rather than defer errors to a downstream system such as a model checker.

The syntax of types τ is shown in Fig. 5; they are standard except for party types, *i.e.* *party* R, which are indexed by role. There are two typing judgments partially defined in Fig. 7 (and fully in the appendix): $\Gamma \vdash_\downarrow e : \langle \tau, \ell\rangle$ for expressions (where each expression e has a type τ as well as a location ℓ) and $\Gamma \vdash G$ for global protocols (denoting that G is well-typed under the assumptions in Γ).

We define a notion of *compatibility* between locations $\ell_1 \sim \ell_2$: two locations are compatible if one of them is *global*, or if they are indexed by the same role R. Intuitively, compatibility captures necessary conditions for state known at one location to be usable at another.

We further extend compatibility to relate locations and types: if a type τ is indexed by the same role as a location ℓ_τ, then for an arbitrary ℓ, $\tau \sim \ell$ if $\ell_\tau \sim \ell$. This implies that τ must be a party type if compatible with any location.

Finally, we assume that the initial environment Γ contains bindings for all role variables R: $\Gamma \vdash_{\downarrow} R : \langle \mathit{set}\ (\mathit{party}\ R), \mathit{global} \rangle$[2], as well as built-in functions. In other words, the initial roles and their members are globally known from the beginning of the execution of the protocol, and constant throughout.

We briefly explain the typing judgments shown, starting with the most basic protocols. Assignment requires both variable v and expression e to be located on the same party. Furthermore, the qualifier q must be a party, drawn from some role R. All locations must be compatible with that of *party* R, which intuitively means that only parties of R can change their own local states via assignments.

In transmission, the sender s and recipient r must firstly be parties. Both must be located somewhere compatible with the *type* (or role) of s: as communication is initiated by the sender, we would like to statically determine whether the sender knows of both *itself* and the intended recipient. All expressions used to construct the message must also be known to the sender.

Example 1. The protocol on the left is an ill-typed specification:

```
1  forall p in P
2    p.s = C;
3    forall x in p.s
4      x→p: m
```

```
5  forall p in P
6    p.s = C;
7    (forall x in p.s
8       p→x : n(x=x);
9       x→p : m)
```

`s` is a dynamic set of parties located on and thus known only to parties of role `P`. The send at line 4 is the cause of the problem: because `x` is drawn from `s`, it is also known only to `p`. However, `x` has type *party* C. It is not possible for a party of role `C` to know if it is in the set `s` in order to initiate the message send. The problem is fixed by having parties of role `P` first communicate with parties from `C` (line 8), similar to the well-typed protocol on the right-hand side above.

We now state the main metatheoretic properties of our type system. Full proofs may be found in the appendix.

Theorem 1 (Progress). *Suppose* $\Gamma \vdash G$. *Then either* $G = \mathit{skip}$ *or there is some* G' *such that* $\langle \Sigma, G \rangle \longrightarrow \langle \Sigma, G' \rangle$.

□

Theorem 2 (Preservation). *If* $\Gamma \vdash G$ *and* $\langle \Sigma, G \rangle \longrightarrow \langle \Sigma, G' \rangle$ *then* $\Gamma \vdash G'$.

[2] In practice, this is accomplished by declaring roles in the concrete syntax of the specification language.

□

Theorem 3 (Type & Location Safety). *Well-typed protocols do not get stuck, and no party in a well-typed protocol uses information that it does not have.*

□

Theorem 3 is the statement of type safety. Our contribution is that of *location safety*: because expressions used by each party are guaranteed to be compatible with each other, it is not possible for a party to use information that it does not have. This is useful, for example, if the specifications are used as inputs to a model checker, where state is usually global and attempts to access information that is not (logically) available either results in a runtime error or something subtly going wrong, compromising protocol correctness.

3.3 Well-formedness

The typing rules of Sec. 3.2 prevent some classes of invalid specifications, by ensuring parties are not able to use information they do not have. Here we describe additional syntactic criteria which ensure that specifications are meaningful.

Disjunction. We require *communication structure* (senders, recipients, and transmissions) on both sides of a disjunction to be identical. Specifications such as `(a→b ∨ a→c)` or `(a→c ∨ b→c)` are examples of violations. While it is possible to define projection for such disjunctions, we avoid doing so as it would require making additional assumptions and rewriting the protocol in possibly-undesired ways (e.g. inserting message sends to tell a party on one side not to wait for an incoming message). Despite these restrictions, message types and values may vary, allowing the internal/external choices of session types to be expressed. Having disjunction as primitive instead of selection and branching also allow the expression of nondeterminism, which is very useful in practical specifications.

4 Local Specifications

This section describes the local specification used subsequently to check for protocol conformance, and introduces the *actions* as an intermediate representation of the protocol which is fed to TLA$^+$ to statically verify the protocol.

4.1 Projection

Given a global protocol G, we *project* it to a *local protocol* L, a simpler language intended to resemble the implementation of a single party. This further bridges the gap between the specification and implementation.

The syntax of a local protocols is given in Fig. 8. Since it is no longer a multiparty language, there are separate send and receive operations, and variables no longer require qualification as they are all local. Additionally, the atomic operations of the protocol (*local actions* A) are factored out so that they may

$$
\begin{array}{lr}
L ::= \mathit{skip} & \text{No-op} \\
\mid A & \text{Local action} \\
\mid L \mid\mid L & \text{Parallel} \\
\mid L \,;\, L & \text{Sequencing} \\
\mid L \vee L & \text{Disjunction} \\
\mid e \Rightarrow^* L & \text{When} \\
\mid \forall v \in R.\ L & \text{Quantifiers}
\end{array}
\qquad
\begin{array}{lr}
A ::= v = e & \text{Assignment} \\
\mid \to r : m(v = e) & \text{Send} \\
\mid s \to : m(v) & \text{Receive}
\end{array}
$$

$$
a(v_1, ...) \triangleq \{pre_L\}\{pre_C\}\ A\ \{post_C\}
$$
$$
M = \{a_1, ...\}
$$

Fig. 8: Local protocols, local actions, actions, monitors

$$
q.v = e \upharpoonright P \triangleq
\begin{cases}
v = e & \text{if } v \downarrow P, e \downarrow P, q = \mathit{self} \\
\mathit{skip} & \text{otherwise}
\end{cases}
$$

$$
e \Rightarrow^* G \upharpoonright P \triangleq
\begin{cases}
e \Rightarrow^* (G \upharpoonright P) & \text{if } e \downarrow P \\
G \upharpoonright P & \text{otherwise}
\end{cases}
$$

$$
s \to r : m(v = \nu) \upharpoonright P \triangleq
\begin{cases}
\to r : m(v); & \\
\quad s \to : m(v{=}e) & \text{if } r{=}s{=}\mathit{self} \\
s \to : m(v{=}e) & \text{if } r{=}\mathit{self} \\
\to r : m(v) & \text{if } s{=}\mathit{self} \\
\mathit{skip} & \text{otherwise}
\end{cases}
$$

$$
\forall n \in ps.\ G \upharpoonright P \triangleq
\begin{cases}
\forall n \in ps.\ (G \upharpoonright P) & \text{if } ps \not: Set(Party\ p) \\
(G[self/n] \upharpoonright P) \mid\mid & \\
\quad \forall n \in (ps \setminus \{self\}).\ (G \upharpoonright P) & \text{if } self \in ps \\
\forall n \in (ps \setminus \{self\}).\ (G \upharpoonright P) & \text{if } self \notin ps
\end{cases}
$$

Fig. 9: Projection (select cases)

be treated separately (Sec. 4.2). The structural congruence relation defined in Sec. 3.1 applies also to local protocols, as they differ only in their atomic actions.

We extend the projection algorithm of Deniélou et al. [13] to support location, state, conditional, disjunction and quantifiers. This algorithm is suitable because it handles *self-sends*, allowing our language to express *multirole* protocols: where parties in a single role send messages to each other. Self-sends are common in distributed protocols: examples include Raft [45], where parties dynamically change *conceptual role* in the course of leader election, and non-blocking atomic commit [47], where participants play a dual role as a coordinator.

Our modified projection function is partially defined in Fig. 9 (a full definition can be found in the appendix). We denote the projection of a global protocol G with respect to a role R and an arbitrary member of that role *self* by $G \upharpoonright P$.

The most subtle case is that of the quantifiers. If the set being quantified over ps is not of the type $Party\ P$, the quantifier is irrelevant to the current party and thus should stay; we project under it. Otherwise, we must check if the party *self* could be in ps; if so, we must account for the possibility of a self-send. We therefore split it into two parallel cases: one for what *self* does, and one for what is *done to it*. The final case is the same as the latter; since *self* is not in ps, we only need to account for what it is done to it by other parties and roles.

Assignments, transmission, and conditionals are removed or simplified depending on whether they pertain to the current party and role. The remaining

cases are purely structural. Given that statements irrelevant to a role are removed and no statements are reordered, intuitively all program-order relations between actions are preserved.

4.2 Actions

While a local protocol bears some resemblance to an implementation, it is not yet clear how to define the correspondence: an implementation contains significantly more detail and complexity than the specification, as well as other functionality unrelated to the protocol being specified.

To further bridge the gap, we decompose each local protocol into a set of *actions*. These resemble the eponymous actions of TLA, in that they are self-contained, atomic transitions that the protocol's transition system makes. Viewing the protocol as an imperative program, actions resemble basic blocks, or chunks of straight-line code.

Actions serve as a high-level intermediate representation of the protocol. They may be translated into TLA$^+$ to have the protocol verified statically, and into a monitor to have the conformance of an implementation checked dynamically. In the latter case, the user additionally provides a mapping between actions and portions of the implementation, so that the monitor and implementation execute in lockstep. We describe the construction of actions at a high level in this section (details in the appendix), then cover how the mapping is defined in Sec. 5.

An *action* a (see Fig. 8) is a local action with explicit pre- and postconditions. We distinguish *logical* and *control* preconditions; postconditions always apply to control state. A may contain free variables, which we interpret as parameters of the action. Actions also have an integer identifier, which we denote by $id(a)$.

We also define a relation $thread(L, t)$ that labels each (sub)protocol L with a *thread identifier* t. This is possible because all concurrency is statically determined. The main thread has the identifier *main*, threads on either side of a parallel composition $L_1 \,||\, L_2$ fresh identifiers t_1, t_2, ..., and identifiers for threads under a quantifier $\forall v \in R.\ L$ are parameterized, e.g. $t_1(v)$.

Example 2. The following specification has 2 thread identifiers, $t_1(c)$ and $t_2(c)$.

```
forall c in C
  c.a = 1 || c.b = 2
```

Given a local protocol, we translate it into a set of actions by encoding all control flow and concurrency constraints as logical and control pre- and postconditions, and turning quantified variables into parameters. This is on some level similar to what the PlusCal [29] translator does for its source language. A major difference in our source language and approach is the conversion of parallel composition, disjunction, and quantifiers into parameterized formulae to be used as control preconditions. These *control predicates* have the following form:

$$CP ::= t(p_1, ...)@ActionId \mid Any(CP_1, ...) \mid All(CP_1, ...) \mid \forall v \in R.\ CP$$
$$ActionId ::= id(a) \mid Start$$

Example 3. To illustrate concretely how control predicates work, we may, like PlusCal, represent the control state of the program using a total mapping $pc : t \to \mathit{ActionId}$.

Control predicates may thus be interpreted as constraints on pc: $t@\mathit{ActionId}$ on a party requires $pc(t) = \mathit{ActionId}$. *Start* represents the control location that all threads are initially at, and is what makes pc total.

The control precondition pre_C of an action a must hold before the execution of a, and the control postcondition $post_C$ must hold after, so concretely the effect of the action must also include updating pc according to $post_C$.

Example 4. To put this in the context of the running example from Sec. 2, the precondition of the action *CSendCommit4* in Fig. 4a, which corresponds to the state change the coordinator effects when it sends a `commit` message to a participant, captures the requirement that all `P` threads of communication must be completed by that point. Therefore the formula is $\forall v \in R.\ \mathit{Any}(t(v)@1, t(v)@2)$, where $t(v)$ is a parameterized thread identifier for the vth thread, and 1 and 2 are identifiers for the actions on either side of the disjunction (the reason *Any* is used) that ends the first phase of the protocol.

The actions in local protocols may be seen intuitively as a graph, where edges denote program-order relations between (parameterized) actions. The graphs in Fig. 4a and Fig. 4b give an overview of how the protocol goes through its phases, though they are useful mainly for user comprehension, as the preconditions encode more granular control constraints.

4.3 Monitors

For brevity, we only briefly introduce the notion of monitor and its associated translation from local protocol, which we term as *monitor synthesis*. Abstractly, a monitor M is simply a set of actions (see Fig. 8).

It is essential that the monitoring approach used is sound, i.e. that monitor violations correctly indicate protocol violations. To ensure this, we provide a soundness result relating the reduction of a local protocol to the execution of the monitor we synthesize from it. $\xrightarrow{M*}\!\!\!\!\!\!\not\;\;$ and $\xrightarrow{L*}\!\!\!\!\!\!\not\;\;$ formalize "getting stuck", denoting that the respective relations do not hold. The full proofs and definitions are given in the appendix.

Theorem 4 (Monitor Synthesis Soundness). *Given* $\mathit{compile}(L) = M$*, if* $\langle \sigma, c, M \rangle \xrightarrow{M*}\!\!\!\!\!\!\not\;\;$*, then* $\langle \sigma, L \rangle \xrightarrow{L*}\!\!\!\!\!\!\not\;\;$.

Proof. We prove the contrapositive: if $\langle \sigma, L \rangle \xrightarrow{L*} \langle \sigma', L', \tau \rangle$, then $\langle \sigma, c, M \rangle \xrightarrow{M*} \langle \sigma', c, \tau \rangle$. In other words, for all traces, if the local protocol does not get stuck, then the monitor does not get stuck. □

Intuitively, Theorem 4 states that the monitor never produces false positives, and that every violating verdict it reaches is an instance of protocol nonconformance. We do not prove completeness, as monitoring is a dynamic analysis

method and is necessarily incomplete – it is always possible for a monitor action *not* to be executed, i.e. if the code which drives it is simply never encountered, so it is not possible to guarantee that every instance of protocol nonconformance results in a monitor violation. Burlò et al. [3] provide a weak-completeness result, requiring the rather strong assumption that there is no dead code. As this is a property not only of programs but executions, we cannot guarantee this.

5 Runtime Verification

Having decomposed a global protocol into sets of actions local to each party, we focus now on the problem of checking that an implementations conforms to a given specification and safety property.

We make use of runtime verification to achieve both goals. As a dynamic analysis method, it is able to observe only individual executions of a system and is thus incomplete. However, it qualifies as a lightweight testing method, may be adopted piecemeal, and benefits from an existing test suite (which an existing implementation would likely have), making it highly compatible with our goals.

5.1 Safety Violations

We build on the work of Bauer et al. [4], who propose LTL_3 , a temporal logic with a three-valued semantics designed for runtime verification.

Classic LTL is interpreted over infinite traces. The traces observed in monitoring, however, come from concrete executions and are necessarily finite. Moreover, full traces are not available in an online context – thus the problem is to determine the truth value of an LTL property given *prefixes* of finite traces. Given such a finite trace prefix u and an LTL_3 formula ϕ, ϕ is *true* if all extensions of u satisfy ϕ (interpreted as a standard LTL formula), *false* if no extensions of u satisfy ϕ, and *inconclusive* otherwise, as it is possible that not enough of the trace has been observed to know if ϕ *cannot* be satisfied or violated.

Given a safety property as an expression e enriched with LTL connectives, we compile it into a monitor in the implementation language. Each non-LTL expression $e_1, e_2, ...$ in e is substituted with a fresh proposition $p_1, p_2, ...$, resulting in a pure LTL formula. From this, we produce a minimized DFA where each state is associated with a truth value, and transitions are dependent on the truth values of $p_1, p_2, ...$ The number of transitions is exponential in the number of propositions, though in practice, the latter is usually small.

We then check if the safety property is *monitorable*. For example, the LTL formula $\Box\Diamond\phi$ is non-monitorable, as given a finite trace prefix, there is no way to extend it to get a truth value other than *inconclusive*. This appears as a DFA with a single *inconclusive* sink state. $e_1, e_2, ...$ are then individually compiled to the implementation language, to be evaluated against its abstract state σ.

We next translate party-local variables into a data structure in the implementation language, giving us a concrete *representation* of its abstract state σ. As in other runtime verification approaches [18], the user must provide two things:

1. An *abstraction function* or *refinement mapping* α, converting the concrete state of a party σ^i into an abstract state σ.
2. Annotations of the implementation's *commit points* (where the implementations abstract state is observed to change) by calling a function *StepS*.

Given an invocation $StepS(\sigma^i)$, the monitor computes $\alpha(\sigma^i) = \sigma$, determines the truth value of each proposition $p_1 = e_1(\sigma), ...$, uses that to trigger a DFA transition, then acts on the truth value of the resulting state. If it is anything other than *inconclusive*, *StepS* returns a value indicating success or failure and disables itself, returning the same value from that point on.

What is notable about this monitoring method is that it is highly transparent and easy to integrate with any programming language, given the simple structure of the generated monitor (an automaton) and annotations (function definition and calls). Additional automation via aspects, effects, or monads may also be layered on top. The user is in control of the size of the automaton and the precise properties being checked (via the LTL property), and the overhead of producing an abstract state (via the abstraction function).

Additionally, this monitor is completely state-based and does not utilize protocol actions at all; minimally, the only inputs required are the safety property and a definition of party-local state. We hope that this makes the method *accessible* and *incrementally applicable* to implementations of any size and complexity. For additional assurance, a user may also check that the actions taken by an implementation conform to those allowed by the protocol; we cover this next.

5.2 Refinement Violations

Refinement checking is the problem of determining if an execution trace of an implementation is equivalent to some execution trace of a specification. If *all* implementation traces are consistent with those of the specification, we say that the implementation *refines* the specification. With a runtime verification method, however, we are necessarily only able to observe *some* implementation traces, and so cannot guarantee refinement. Therefore we focus instead on the problem of detecting refinement *violations*.

Given the abstraction function α from Sec. 5.1 and a set of actions $\{a_1, a_2, ...\}$, we generate a monitor. With all the machinery defined in Sec. 4.2 and Sec. 4.3, monitors are straightforward to compile: we generate runtime analogues of action identifiers $id(a_i)$ and parties p_j in the implementation language, and represent the control state of the specification via a variable *pc* as in Example 3.

We then enrich commit point annotations with a function *StepA*. The intent is for the user to define the correspondence between a local action and a portion of implementation code by calling *StepA* with the appropriate arguments. $StepA(\sigma^i, id(a_i), p_1, p_2, ...)$ checks the logical precondition pre_L of a_i against $\sigma = \alpha(\sigma^i)$, checks the control precondition pre_C of a_i against $pc[t]$ (where $thread(a_i, t)$, interpreting the local action A of a_i as a local protocol), then modifies *pc* using its control postcondition $post_C$.

In this way, we essentially transition enough of the specification in lockstep with the implementation at runtime to determine if the preconditions of each

action hold. A precondition not holding is tantamount to a violation of a local protocol's program-order relation, where the implementation would have taken a step which the specification never could have taken. In other words, the current execution trace of the implementation cannot be one of the specification's, witnessing a refinement violation.

For a practical monitoring method, there is the additional consideration that actions should not be too granular: ideally they would correspond to high-level protocol operations, like a user would define in TLA$^+$, and correspond to operations the implementation itself performs.

This is difficult to define precisely and generically for all protocols *and* implementations, so as a first approximation, our implementation fuses contiguous actions on the same thread, creating compound actions of one of the following shapes: a sequence of assignments, a receive followed by a sequence of assignments, or a send. In our experiments these shapes worked well for annotating a number of real-world protocol implementations. An example of instrumented code may be found in the appendix.

6 Implementation and Evaluation

We prototyped our approach and used it to specify and verify a variety of realistic distributed protocol implementations. The general approach to using the prototype is as follows. First, a protocol specification is written, including safety properties. These specifications are type-checked and decomposed into actions. From there both a TLA$^+$ model and a monitor implementation in Go are generated. The model may be given to TLC, the TLA$^+$ model checker, to be validated, while the monitor supports checking for safety property and refinement violations. We chose Go because of its popularity for the sort of infrastructure work that involves distributed protocols, but our approach is language-agnostic: only the final stage of monitor compilation instantiates it for a particular language.

Evaluation. We evaluate our approach by answering the following questions, divided along two broad axes:

1. What are the classes of protocols our specification language excels at expressing? How concise are our renderings of this class of protocols compared to existing specification languages?
2. How scalable (in terms of effort required) is the runtime verification approach on existing codebases? How much overhead is imposed at runtime?

Expressiveness. Our specification language is able to express multiparty communication with branching and synchronization, but without recursion. This lends itself well to the expression of simple consensus and commit protocols. We compare the relative sizes of specifications of these protocols in Tab. 1.

We chose DistAlgo [33] and TLA$^+$ because they have similar goals, support a high level of abstraction (focusing exclusively on protocol logic, as opposed to

Protocol	Ours	DistAlgo	TLA$^+$
Two-phase commit [31]	18	-	66
Two-phase commit (synchronous)	20	55	-
Single-decree Paxos [30]	31	73	85
Non-blocking atomic commit [47]	21	-	96

Table 1: Relative specification sizes (LoC)

details of transport and networking, as e.g. IronFleet[21] or Mace [26] do) and are representative of different specification paradigms.

TLA$^+$ renders a protocol as a transition system in a first-order temporal logic. Message sends and receives are typically expressed as separate transitions to model asynchrony. This may not always be the case, as TLA$^+$ is extremely flexible and abstract, making communication structure emergent: it is possible to not model participants or parties at all if so desired.

DistAlgo instead renders protocols as imperative Python-like programs. This has the benefit that specifications are directly executable. Parties are represented as separate classes, and message receives by handler functions. DistAlgo also uses quantification over message histories, which removes the bookkeeping required to express synchronization conditions, so it is more succinct than, say, raw Python.

Our language instead uses a choreographic paradigm, emphasizing communication as the primary structuring mechanism and deriving the separate roles via projection. This has the benefit of making specifications much shorter, with the tradeoff that the individual role each participant plays in the overall protocol is less clear. However, the overall protocol is clearer; comparing the English description of single-decree Paxos with our specification (see appendix), there is a direct mapping between text and specification when read in order. This is not a coincidence as natural language descriptions are often presented in a causal fashion coinciding with the multiparty meaning of sequential composition.

We are aware that LoC is at best a proxy for succinctness and clarity, as these languages make different tradeoffs. The point of this is to show that choreographies are a promising paradigm for distributed protocols.

Runtime Verification. To demonstrate the applicability of our approach, we instrumented a number of real-world projects to show their conformance to a simple protocol, two-phase commit. The projects were taken from GitHub and selected simply for being reasonably complete, runnable implementations of the protocol in Go (i.e. mentioning the name of the protocol somewhere in the readme or tags). More importantly, the specification was not written with them in mind, and the projects were not designed for runtime verification.

We noted the size of each project (in LoC; not including the combined 678 lines of generated monitor code), how much effort it took (in hours spent by a user knowledgeable in both the implementation and specification languages) to build and run them, never having seen them before (*comprehension time*), and how much time it then took to instrument each project and check for conformance in

Project	LoC	Compr. (hrs)	Instr. (hrs)	Overh. (2)	Overh. (4)	Overh. (6)
prevosis/distributed-transactions	1205	2	2	0.09%	0.11%	0.10%
vadiminshakov/committer	2430	1	1	1.54%	1.65%	1.45%
ianobermiller/gotwopc	2271	1	1	0.08%	0.13%	0.17%

Table 2: Instrumentation effort

a representative execution (*instrumentation time*), and the amount of runtime overhead incurred for different numbers of replicas. The results are in Tab. 2.

The manual effort required was low – on the order of hours, and scaling with specification size, not implementation size. Given a project maintainer's understanding of the implementation, only the instrumentation time would factor.

Our experiments to measure runtime overhead were carried out on a 2020 MacBook Pro with a 2.3 GHz CPU and 16 GB RAM, with Go 1.16.3.

All the projects were replicated key-value stores with HTTP or (g)RPC interfaces. Each experiment was carried out by starting a master node and some number of replicas (reflecting realistic cluster sizes), then starting a client program which issued 100 requests to write different keys (triggering each key-value store's commit protocol) in a blocking manner. Measurements were done by manually instrumenting the monitor and the relevant parts of the implementation with Go's monotonic clock functions. The overhead figures in Tab. 2 represent the percentage of time *across all replicas* spent in monitor code over the amount of time the client waited. We ran each experiment 5 times and report the average. The runtime overhead was extremely low in general ($< 2\%$), supporting the claim that our method is lightweight.

As to its effectiveness, we discovered a refinement violation in the first project, where the coordinator mishandles an error from a participant and commits the transaction anyway, despite not getting a response to its `prepare` message. This was caused by a misplaced `continue` in a loop which allowed a `commit` message to be a sent to a participant *without* it having first seen a `prepared` message.

7 Related Work

Our work occupies a particular niche: it targets the problem of bridging a specification with an *existing* implementation, if the method of specification may be designed anew. It is a direct answer to the question of what to do after an "executable pseudocode" [38] specification has been written for a system.

Protocol Specifications. Protocol specification has a long history. Perhaps the most well-known way to specify a protocol is semantically: as a transition system or automaton [28]. Many contemporary specification languages, such as TLA$^+$ [28], Mace [26], Statecall Policy Language [35], Distal [5], Ivy [46], and P/ModP [17] use this representation. It is also a good representation for an embedding of a protocol in a proof assistant [52,50]. While such approaches

generally offer strong guarantees, the annotation overhead they require is too high for these methods to be part of a Continuous Integration platform.

Another family of specification languages include choreographic languages, which describe interactions globally and feature program structuring mechanisms. Notable representatives include multiparty session types [22,13], and languages descended from them such as Scribble [54] and Choral [19].

A final method is to represent a protocol directly as an imperative but high-level program, in the spirit of "executable pseudocode". DistAlgo [33] is a framework in this category, as an extension of Python with constructs for querying message history directly in programs.

Our work sits at the intersection of all of these approaches. We utilize a choreographic approach with some elements of imperative and concurrent programming, ultimately translating down to a transition system semantics.

The final point may be evocative of PlusCal [29] and its distributed variant [2], but we specialize further for the domain of distributed protocols by being more high-level and abstract, doing away with channels and using an actor-like model with processes identified only by party.

Session Types. Several works address the monitoring of protocols described by session types [39,42,41,6]. An early informal approach indicates that deriving monitors from types is feasible and outlines the benefits of doing so [10]. Next, [7] builds up on the ideas of [10] formally proving properties such as local/-global safety, and session fidelity. However, the results in [7], later implemented in [43,23], only hold for safe transport network, while we consider the more realistic scenario where messages may be dropped, duplicated, reordered, or arbitrarily delayed. Similar to our work, [3] studies the formalization of session monitorability in the presence of unreliable network, proving the soundness of monitor synthesis. Different than our work, [3] only handles binary session types thus limiting the kind of protocols it can handle.

The idea of role-parametric session types has been investigated before in [9,16] where roles are parameterized by indices—unlike our work which *iterates over role sets* supporting therefore more concise protocol descriptions.

The close relation between communicating automata and asynchronous multiparty session types was investigated in [14] which shows that session types are in essence a syntactical representation of communicating automata. This relation inspired numerous works which derive communication API from multiparty session types [49,40,24], synthesize the global protocol from a collection of communicating automata [15], or verify properties of the communicating automata [32,44]. Although tools build on these works guarantees that communicating automata representing session types interact correctly, none has investigated the benefits of introducing TLA$^+$ in the workflow to further explore the expressiveness of the enforced properties.

Runtime Verification. Several specification languages for distributed protocols offer some support for verification via model checking or testing. We distinguish those which require clean-room implementations using a new programming model [26,33] from those which target existing systems, like ours.

Scribble [54] supports conformance testing of implementations by generating classes corresponding to *endpoint FSM* of a projected protocol to enforce linear usage of channels. Statecall Policy Language [35] is a specification language for protocols expressed as NFAs and supports compilation to OCaml for runtime verification. APSL [51] tests for protocol conformance. It provides support for random generation of messages and traversing protocol state spaces modeled as transition systems. Our approach has the same goals as all of these, but is based on TLA as a formalism and supports arbitrary monitorable safety properties in LTL as well as refinement violations. We do not generate tests but take a lightweight approach to integration with an existing implementation.

We build on the LTL_3 semantics [4] for trace monitoring. Mostafa et al. [37] propose a similar approach that uses LTL_3 for decentralized runtime verification. Our approach instead restricts the safety property so that it is observable at some party. As an example, consensus in single-decree Paxos is technically achieved the moment a majority of acceptors accept a value. However, in a realistic system, no one would be able to *act* on this decentralized decision until at least one learner is made aware of it. We thus restrict the safety property so that it can be expressed with the information available to learners. This is broadly similar to instrumenting the system with auxiliary state – in this case, we extend the protocol with auxiliary *actions* to centralize the decision. We hope to integrate some of the insights of their work in future.

8 Conclusion

We propose a new multiparty specification language for distributed protocols in order to facilitate protocol specification and check conformance with an existing implementation using runtime verification. This gives us a practical and incremental approach for improving the robustness of protocol implementations.

We further plan to investigate classes of protocols which are currently difficult to express, such as role-parametric leader election protocols [8] and protocols with recursion and cyclic action graphs. Fault tolerance is another consideration which we have not adequately addressed yet. We also want to broaden the scope of LTL properties that our runtime verification system can check, including decentralized ones [37] and those which are currently not monitorable [36].

References

1. runtime: delete proc.p. https://github.com/golang/go/commit/0e027fca428a893ceeb49e4f833f9a923c5f201d, accessed: 2021-05-27
2. Alkayed, H., Cirstea, H., Merz, S.: An extension of pluscal for modeling distributed algorithms. In: TLA+ Community Meeting (2020)
3. Bartolo Burlò, C., Francalanza, A., Scalas, A.: On the Monitorability of Session Types, in Theory and Practice. In: 35th European Conference on Object-Oriented Programming (ECOOP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2021)
4. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. Transactions on Software Engineering and Methodology (TOSEM) pp. 1–64 (2011)

5. Biely, M., Delgado, P., Milosevic, Z., Schiper, A.: Distal: A framework for implementing fault-tolerant distributed algorithms. In: International Conference on Dependable Systems and Networks (DSN). pp. 1–8. IEEE (2013)
6. Bocchi, L., Chen, T.C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. In: Formal Techniques for Distributed Systems, pp. 50–65. Springer (2013)
7. Bocchi, L., Chen, T.C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. Theoretical Computer Science **669**, 33–58 (2017). https://doi.org/https://doi.org/10.1016/j.tcs.2017.02.009
8. Castro, D., Hu, R., Jongmans, S.S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in Go. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
9. Castro, D., Hu, R., Jongmans, S.S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: Statically-typed endpoint apis for dynamically-instantiated communication structures **3**(POPL) (2019)
10. Chen, T.C., Bocchi, L., Deniélou, P.M., Honda, K., Yoshida, N.: Asynchronous distributed monitoring for multiparty session enforcement. vol. 7173, pp. 25–45 (06 2011)
11. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: International Symposium on Formal Methods for Components and Objects. pp. 266–296. Springer (2006)
12. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA+ proofs. In: International Symposium on Formal Methods (2012)
13. Deniélou, P.M., Yoshida, N.: Dynamic multirole session types. In: ACM SIGPLAN-SIGACT Symposium on Principles of programming languages. pp. 435–446 (2011)
14. Deniélou, P.M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) Programming Languages and Systems (2012)
15. Deniélou, P.M., Yoshida, N.: Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In: International Colloquium on Automata, Languages, and Programming. pp. 174–186. Springer (2013)
16. Deniélou, P.M., Yoshida, N., Bejleri, A., Hu, R.: Parameterised Multiparty Session Types. Logical Methods in Computer Science **Volume 8, Issue 4** (2012). https://doi.org/10.2168/LMCS-8(4:6)2012, https://lmcs.episciences.org/924
17. Desai, A., Phanishayee, A., Qadeer, S., Seshia, S.A.: Compositional programming and testing of dynamic distributed systems (OOPSLA), 1–30 (2018)
18. Elmas, T., Tasiran, S., Qadeer, S.: Vyrd: verifying concurrent programs by runtime refinement-violation detection. ACM SIGPLAN Notices **40**(6), 27–37 (2005)
19. Giallorenzo, S., Montesi, F., Peressotti, M.: Choreographies as objects. arXiv
20. v. Gleissenthall, K., Kıcı, R.G., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony: synchronous verification of asynchronous distributed programs. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
21. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 1–17 (2015)
22. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284 (2008)
23. Hu, R., Neykova, R., Yoshida, N., Demangeon, R., Honda, K.: Practical interruptible conversations. In: Runtime Verification (2013)
24. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: International Conference on Fundamental Approaches to Software Engineering. pp. 116–133. Springer (2017)

25. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming **28** (2018)
26. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: language support for building distributed systems. ACM Sigplan Notices pp. 179–188 (2007)
27. Konnov, I., Kukovec, J., Tran, T.H.: Tla+ model checking made symbolic. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–30 (2019)
28. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3), 872–923 (1994)
29. Lamport, L.: The pluscal algorithm language. In: International Colloquium on Theoretical Aspects of Computing. pp. 36–60. Springer (2009)
30. Lamport, L., et al.: Paxos made simple. ACM Sigact News **32**(4), 18–25 (2001)
31. Lampson, B., Sturgis, H.E.: Crash recovery in a distributed data storage system (1979)
32. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2019)
33. Liu, Y.A., Stoller, S.D., Lin, B., Gorbovitski, M.: From clarity to efficiency for distributed algorithms. pp. 395–410. No. OOPSLA (2012)
34. Ma, H., Goel, A., Jeannin, J.B., Kapritsos, M., Kasikci, B., Sakallah, K.A.: I4: incremental inference of inductive invariants for verification of distributed protocols. In: SOSP. pp. 370–384 (2019)
35. Madhavapeddy, A.: Combining static model checking with dynamic enforcement using the statecall policy language. In: International Conference on Formal Engineering Methods. pp. 446–465. Springer (2009)
36. Mascle, C., Neider, D., Schwenger, M., Tabuada, P., Weinert, A., Zimmermann, M.: From LTL to rLTL monitoring: improved monitorability through robust semantics. In: HSCC. pp. 1–12 (2020)
37. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of ltl specifications in distributed systems. In: 2015 IEEE International Parallel and Distributed Processing Symposium. pp. 494–503. IEEE (2015)
38. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Communications of the ACM pp. 66–73 (2015)
39. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Aspects of Computing **29**(5), 877–910 (2017)
40. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A Session Type Provider: Compile-Time API Generation of Distributed Protocols with Refinements in F#. In: Proceedings of the 27th International Conference on Compiler Construction. Association for Computing Machinery, New York, NY, USA (2018)
41. Neykova, R., Yoshida, N.: Multiparty session actors. In: International Conference on Coordination Languages and Models. pp. 131–146. Springer (2014)
42. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: Proceedings of the 26th International Conference on Compiler Construction. pp. 98–108 (2017)
43. Neykova, R., Yoshida, N., Hu, R.: Spy: Local verification of global protocols. In: Runtime Verification. pp. 358–363. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
44. Ng, N., Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 174–184 (2016)

45. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: USENIX. pp. 305–319 (2014)

46. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: PLDI. pp. 614–630 (2016)

47. Raynal, M.: A case study of agreement problems in distributed systems: non-blocking atomic commitment. In: HASE. pp. 209–214 (1997)

48. Reid, A., Church, L., Flur, S., de Haas, S., Johnson, M., Laurie, B.: Towards making formal methods normal: meeting developers where they are. arXiv:2010.16345

49. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: 31st European Conference on Object-Oriented Programming (ECOOP 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)

50. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols **2**(POPL), 1–30 (2017)

51. Tervoort, T., Prasetya, I.: Modeling and testing implementations of protocols with complex messages. arXiv preprint arXiv:1804.03927 (2018)

52. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI. pp. 357–368 (2015)

53. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.: Planning for change in a formal verification of the raft consensus protocol. In: CPP. pp. 154–165 (2016)

54. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The Scribble protocol language. In: International Symposium on Trustworthy Global Computing. pp. 22–41 (2013)

55. Yu, Y., Manolios, P., Lamport, L.: Model checking tla+ specifications. In: Correct Hardware Design and Verification Methods. pp. 54–66. Springer (1999)

9 Appendix

9.1 Operational Semantics

Note the use of the *in-progress transmission* $\square \to r : m(v = \nu)$ to model asynchrony, which appears in the course of reduction (in G-SEND and G-RECEIVE) but is not part of the abstract syntax (and so is absent from the definition of G in Fig. 5).

A variation on this method of modeling asynchrony appears in the original MPAST paper [22], where the message soup is modeled as a separate *message queue* process. We take the view that asynchronous interactions with the message soup are an implementation detail, and so consider it an auxiliary protocol state and de-emphasize it in the abstract syntax.

G-FORALL

$$\frac{p = \{p_1, p_2, ...\}}{\langle \Sigma, \forall v \in p.\ G\rangle \longrightarrow \langle \Sigma, G[p_1/v] \mid\mid G[p_2/v] \mid\mid ...\rangle}$$

G-ASSIGN

$$\frac{v, e \downarrow party\ R \qquad q \in R \qquad \Sigma(q) = \sigma \qquad \langle \sigma, e\rangle \Downarrow \nu \qquad \sigma' = \sigma[\nu/v]}{\langle \Sigma, q.v = e\rangle \longrightarrow \langle \Sigma[\sigma'/q], skip\rangle}$$

G-SEND

$$\frac{s \in R \qquad s, r, e \downarrow party\ R \qquad M' = M \cup \{\!\{\langle s, r, m(v = \nu)\rangle\}\!\}}{\langle \Sigma, s \to r : m(v = \nu)\rangle \longrightarrow \langle \Sigma[M'/M], \square \to r : m(v = \nu)\rangle}$$

G-RECEIVE

$$\frac{M = M' \cup \{\!\{\langle s, r, m(v = \nu)\rangle\}\!\}}{\langle \Sigma, \square \to r : m(v = \nu)\rangle \longrightarrow \langle \Sigma[M'/M], r.v = e\rangle}$$

G-RECEIVE-BLOCK

$$\frac{}{\langle \Sigma, s \to r : m(v = \nu)\rangle \longrightarrow \langle \Sigma, s \to r : m(v = \nu)\rangle}$$

G-SEQ

$$\frac{\langle \Sigma, G_1\rangle \longrightarrow \langle \Sigma', G_1'\rangle}{\langle \Sigma, G_1\ ;\ G_2\rangle \longrightarrow \langle \Sigma', G_1'\ ;\ G_2\rangle}$$

G-PAR

$$\frac{\langle \Sigma, G_1\rangle \longrightarrow \langle \Sigma', G_1'\rangle}{\langle \Sigma, G_1 \mid\mid G_2\rangle \longrightarrow \langle \Sigma', G_1' \mid\mid G_2\rangle}$$

G-DISJ

$$\frac{\langle \Sigma, G_1\rangle \longrightarrow \langle \Sigma', G_1'\rangle}{\langle \Sigma, G_1 \vee G_2\rangle \longrightarrow \langle \Sigma', G_1'\rangle}$$

G-WHEN-T

$$\frac{e \downarrow \ell \qquad \ell \sim party\ R \qquad p \in R \qquad \Sigma(p) = \sigma \qquad \langle \sigma, e\rangle \Downarrow true}{\langle \Sigma, e \Rightarrow^* G\rangle \longrightarrow \langle \Sigma, G\rangle}$$

G-WHEN-F

$$\frac{e \downarrow \ell \qquad \ell \sim party\ R \qquad p \in R \qquad \Sigma(p) = \sigma \qquad \langle \sigma, e\rangle \Downarrow false}{\langle \Sigma, e \Rightarrow^* G\rangle \longrightarrow \langle \Sigma, e \Rightarrow^* G\rangle}$$

9.2 Expression Evaluation

Fig. 10 shows a standard call-by-value big-step evaluation relation for pure expressions, as mentioned in Sec. 3.1.

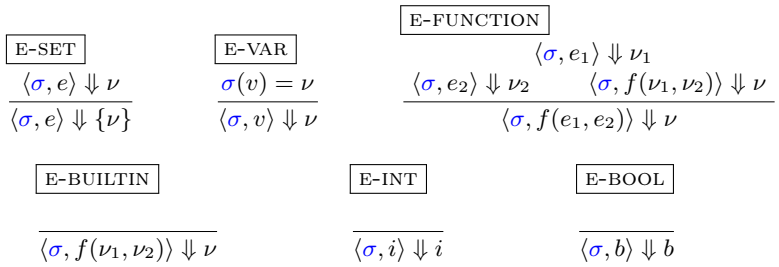


Fig. 10: Expression evaluation

9.3 Type System

See Fig. 11.

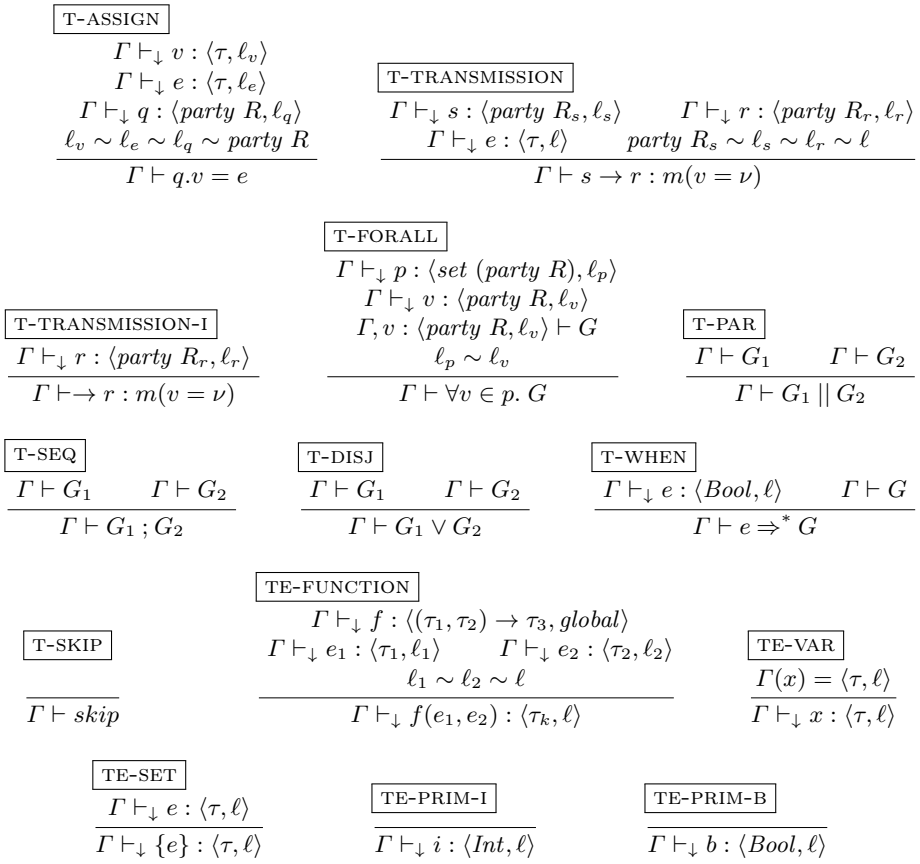


Fig. 11: Type system

9.4 Type safety

We first state some properties of $\Downarrow$, which are equally standard.

Theorem 5 (Expression Progress). *Suppose $\Gamma \vdash_{\downarrow} e : \langle \tau, \ell \rangle$. Then either e is a value or there is some ν such that $\langle \sigma, e \rangle \Downarrow \nu$.*

Proof. By induction on a derivation of $\Gamma \vdash_{\downarrow} e : \langle \tau, \ell \rangle$. □

Theorem 6 (Expression Preservation). *If $\Gamma \vdash_{\downarrow} e : \langle \tau, \ell \rangle$ and $\langle \sigma, e \rangle \Downarrow \nu$ then $\Gamma \vdash_{\downarrow} \nu : \langle \tau, \ell \rangle$.*

Proof. By induction on a derivation of $\Gamma \vdash_{\downarrow} e : \langle \tau, \ell \rangle$. □

The only noteworthy addition is that of locations to the typing context, however they do not cause trouble as each rule has premises constraining them to be compatible.

Next, we state and prove properties of global protocol reduction.

Theorem 1 (Progress). *Suppose $\Gamma \vdash G$. Then either $G = skip$ or there is some G' such that $\langle \Sigma, G \rangle \longrightarrow \langle \Sigma, G' \rangle$.*

Proof. By induction on a derivation of $\Gamma \vdash G$.
Case T-SKIP: immediate as $G = skip$.
Case T-ASSIGN: q, v, and e are all well-typed and located in parties of role R. The next state therefore sees the local state of q updated with v bound to some value e'.
Case T-TRANSMISSION: s are r are known to be parties, and both along with the expressions used to construct the message are known to the sender. This is sufficient to determine that the next state will have the message moved to the soup and the transmission replaced with an in-progress one by G-SEND.
Case T-TRANSMISSION-I: similar to T-TRANSMISSION except that the only requirement is for r to be a party. The next step is derived by G-RECEIVE and has similarly few constraints, as the only way to get an in-progress transmission is via a send, which has stronger premises.
Case T-WHEN: e is a boolean value and has some location ℓ. We know that ℓ is either *global* or *party* R' by definition, and from this that it is compatible with some location of the form *party* R. In other words, there exists some party $p \in R$ which may access all of the conditional expression. This results the global protocol G taking a step by either G-WHEN-T or G-WHEN-F.
Case T-FORALL: given that p is a party and v is a role with compatible locations, applying the induction hypothesis lets us conclude that the next state is the parallel composition of global protocols where one has taken a step.
Cases T-PAR, T-SEQ, T-DISJ: similarly straightforward applications of the induction hypothesis. □

Theorem 2 (Preservation). *If $\Gamma \vdash G$ and $\langle \Sigma, G \rangle \longrightarrow \langle \Sigma, G' \rangle$ then $\Gamma \vdash G'$.*

Proof. By induction on a derivation of $\Gamma \vdash G$.
Case T-SKIP: vacuously true as *skip* does not take a step.
Case T-ASSIGN: G' is *skip*, which is well-typed by T-SKIP.
Case T-TRANSMISSION: G' is an in-progress transmission, which is well-typed by T-TRANSMISSION-I.
Case T-TRANSMISSION-I: G' is an assignment, which is well-typed by $t-assign$.
Case T-WHEN: from the induction hypothesis, we know that reduction of G is type-preserving. It suffices to show that evaluation after adding the guard e preserves types. Applying G-WHEN-T, this is immediate from the induction hypothesis. Applying G-WHEN-F, we trivially have preservation because $G = G'$.
Case T-FORALL: G' is a parallel composition of global protocols. By the induction hypothesis, reduction on each one is type-preserving. It suffices to show that parallel composition preserves types, which we have by applying T-PAR.
Cases T-PAR, T-SEQ, T-DISJ: similarly straightforward applications of the induction hypothesis. □

Theorem 3 (Type & Location Safety). *Well-typed protocols do not get stuck, and no party in a well-typed protocol uses information that it does not have.*

Proof. Corollary of Theorem 1 and Theorem 2. □

9.5 Projection

See Fig. 12.

$$q.v = e \upharpoonright P \triangleq \begin{cases} v = e & \text{if } v \downarrow P, e \downarrow P, q = \mathit{self} \\ \mathit{skip} & \text{otherwise} \end{cases}$$

$$e \Rightarrow^* G \upharpoonright P \triangleq \begin{cases} e \Rightarrow^* (G \upharpoonright P) & \text{if } e \downarrow P \\ G \upharpoonright P & \text{otherwise} \end{cases}$$

$$G_1 \,;\, G_2 \upharpoonright P \triangleq (G_1 \upharpoonright P) \,;\, (G_2 \upharpoonright P)$$
$$G_1 \vee G_2 \upharpoonright P \triangleq (G_1 \upharpoonright P) \vee (G_2 \upharpoonright P)$$
$$G_1 \mid\mid G_2 \upharpoonright P \triangleq (G_1 \upharpoonright P) \mid\mid (G_2 \upharpoonright P)$$

$$s \to r : m(v = \nu) \upharpoonright P \triangleq \begin{cases} \to r : m(v); & \\ \quad s \to : m(v = \nu) & \text{if } r = s = \mathit{self} \\ s \to : m(v = \nu) & \text{if } r = \mathit{self} \\ \to r : m(v) & \text{if } s = \mathit{self} \\ \mathit{skip} & \text{otherwise} \end{cases}$$

$$\forall n \in ps.\ G \upharpoonright P \triangleq \begin{cases} \forall n \in ps.\ (G \upharpoonright P) & \text{if } ps \not\downarrow \mathit{Set}(\mathit{Party}\ p) \\ (G[\mathit{self}/n] \upharpoonright P) \mid\mid & \\ \quad \forall n \in (ps \setminus \{\mathit{self}\}).\ (G \upharpoonright P) & \text{if } \mathit{self} \in ps \\ \forall n \in (ps \setminus \{\mathit{self}\}).\ (G \upharpoonright P) & \text{if } \mathit{self} \notin ps \end{cases}$$

Fig. 12: Projection

9.6 Example of Instrumented Code

The following snippet of code comes from one of the instrumented projects in Sec. 6. We simply invoke the `Step` method as the implementation sends and receives messages (in this case the *prepare* and *abort* messages from two-phase commit), so the monitor and implementation run in lockstep.

```
// Send out all mutate requests in parallel. If any abort, send on the channel.
// Channel must be buffered to allow the non-blocking read in the switch.
shouldAbort := make(chan int, m.replicaCount)
log.Println("Master."+action+"␣asking␣replicas␣to␣"+action+"␣tx:", txId, "key:", key)
m.forEachReplica(func(i int, r *ReplicaClient) {
  pid := strconv.Itoa(i)
  success, err := f(r, txId, i, getReplicaDeath(replicaDeaths, i))
  if err := m.monitor.Step(m.abstract(false), rvc.CSendPrepare8, pid); err != nil {
    log.Printf("%v\n", err)
  }
  if err != nil {
    log.Println("Master."+action+"␣r.Try"+action+":", err)
  }
  if success == nil || !*success {
    if err := m.monitor.Step(m.abstract(false), rvc.CReceiveAbort10, pid); err != nil {
      log.Printf("%v\n", err)
    }
    shouldAbort <- 1
  } else {
    if err := m.monitor.Step(m.abstract(false), rvc.CReceivePrepared9, pid); err != nil {
      log.Printf("%v\n", err)
    }
  }
})
```

9.7 From Local Protocols to Actions

In this section we describe in detail how local protocols are decomposed into actions. We first recall a few prerequisites defined in Sec. 4.

- A *local action* A (Fig. 8) is an atomic operation that a local protocol can perform: a send, receive, or an assignment.
- An *action* a, denoted $\{pre_L\}\{pre_C\}\ A\ \{post_C\}$, is a local action with explicit pre- and postconditions. Actions also have an integer identifier, which we denote by $id(a)$.
- We also define a relation $thread(L, t)$ that labels each (sub)protocol L with a *thread identifier* t.

We now define the function $\xrightarrow{D}$ in Fig. 13, which transforms a local protocol L, using an environment of parameters and logical preconditions, into a set of actions $M = \{a_1, ...\}$ and a control postcondition $post_C$. We denote its invocation by $\langle Params, LP, L\rangle \xrightarrow{D} \langle\{a_1, ...\}, post_C\rangle$.

We first explain the base case for local actions. They create an action a with the given parameters and logical precondition. The control precondition of this action in isolation is $t@Start$, and the postcondition is met when the thread the action is on has control location $i = id(a)$, the fresh identifier of the action.

The D-SEQ rule recursively decomposes two local protocols, then updates the control precondition of the right one to reflect that they appear in sequence. We write e.g. $a.pre_C$ for the control precondition of action a, and use $a.pre_C := c$ to denote updating it to c; this applies analogously for other fields. $b\ ?\ a_1 : a_2$ has its usual meaning as a conditional expression.

The initial actions of the left one also gain the logical precondition LP – other actions do not as their side effects may invalidate LP, something we do not currently reason about (due to the overlap with checking safety properties in Sec. 5.1). The postcondition of the entire sequence is that of the right local protocol.

The D-PAR and D-DISJ rules are dual, producing the postcondition that either *All* or *Any* of their branches are taken. D-FORALL contributes parameters and produces a postcondition expressing that all threads being quantified over should complete. D-IF and D-WHEN contribute logical preconditions.

Actions are straightforward to map to TLA actions, and so might be rendered in TLA$^+$ as (with comments as placeholders):

$$
\begin{aligned}
&CSendPrepare(*\ id\ *)(self, (*\ p, ...\ *)) \triangleq \\
&\quad \wedge\ (*\ pre_L \wedge pre_C\ *) \\
&\quad \wedge\ (*\ A\ *) \\
&\quad \wedge\ (*\ post_C\ *)
\end{aligned}
$$

We may also translate this intermediate representation of the protocol into an implementation language to be used as a monitor. Informally, a monitor is

a set of actions and enough concrete state to check that all preconditions hold. We give monitors a more formal treatment in Sec. 5.1 and Sec. 9.8.

$$\boxed{\text{D-LOCAL-ACTION}}\quad \frac{i = id(a) \qquad fv(A) \subseteq Params \qquad thread(A, t) \qquad a = \{LP\}\{t@Start\}\ A\ \{t@i\}}{\langle Params, LP, A\rangle \xrightarrow{D} \langle \{a\}, t@id\rangle}$$

$$\boxed{\text{D-SKIP}}\quad \frac{}{\langle Params, LP, skip\rangle \xrightarrow{D} \langle \{\}, t@Start\rangle}$$

$$\boxed{\text{D-SEQ}}\quad \frac{\langle Params, true, L_1\rangle \xrightarrow{D} \langle M_1, CP_1\rangle \qquad \langle Params, true, L_2\rangle \xrightarrow{D} \langle M_2, CP_2\rangle \qquad M_1' = \{a.pre_C = t@Start\ ?\ a.pre_L := LP : a \mid a \in M_1\} \qquad M_2' = \{a.pre_C = t@Start\ ?\ a.pre_C := CP_1 : a \mid a \in M_2\}}{\langle Params, LP, L_1\ ;\ L_2\rangle \xrightarrow{D} \langle M_1' \cup M_2', CP_2\rangle}$$

$$\boxed{\text{D-PAR}}\quad \frac{\langle Params, LP, L_1\rangle \xrightarrow{D} \langle M_1, CP_1\rangle \qquad \langle Params, LP, L_2\rangle \xrightarrow{D} \langle M_2, CP_2\rangle}{\langle Params, LP, L_1 \mid\mid L_2\rangle \xrightarrow{D} \langle M_1 \cup M_2, All(CP_1, CP_2)\rangle}$$

$$\boxed{\text{D-DISJ}}\quad \frac{\langle Params, LP, L_1\rangle \xrightarrow{D} \langle M_1, CP_1\rangle \qquad \langle Params, LP, L_2\rangle \xrightarrow{D} \langle M_2, CP_2\rangle}{\langle Params, LP, L_1 \vee L_2\rangle \xrightarrow{D} \langle M_1 \cup M_2, Any(CP_1, CP_2)\rangle}$$

$$\boxed{\text{D-FORALL}}\quad \frac{\langle Params \cup \{v\}, LP, L\rangle \xrightarrow{D} \langle M, CP\rangle}{\langle Params, LP, \forall v \in p.\ L\rangle \xrightarrow{D} \langle M, \forall v \in p.\ CP\rangle}$$

$$\boxed{\text{D-WHEN}}\quad \frac{\langle Params, LP \wedge e, L\rangle \xrightarrow{D} \langle M, CP\rangle}{\langle Params, LP, e \Rightarrow^* L\rangle \xrightarrow{D} \langle M, CP\rangle}$$

Fig. 13: Decomposing a local protocol into actions

9.8 Soundness of Monitor Synthesis

Our work addresses the problem of checking that an implementation conforms to a specified protocol via monitoring. It is thus important that the monitoring approach used is sound, i.e. that monitor violations correctly indicate protocol violations. To ensure this, we prove a soundness result relating the reduction of a local protocol to the execution of the monitor we synthesize from it.

The syntax of local protocols is given in Fig. 8. When a local protocol performs a reduction, it conceptually produces an *event* E by sending or receiving messages, or modifying its local state. The *Skip* event is produced by internal reductions that don't produce any observable side effects, reminiscent of *stuttering* in TLA.

$$
\begin{array}{ll}
E ::= & \textbf{Event} \\
\mid & \mathit{Sent}\ m(v = \nu) \\
\mid & \mathit{Received}\ m(v = \nu) \\
\mid & \mathit{Assigned}\ v = \nu \\
\mid & \mathit{Skip}
\end{array}
$$

A sequence of reductions results in a *trace* $\tau = E_1 E_2 ...$, a sequence of events. The semantics of a local protocol is defined by the set of traces that may be observed in the course of its reduction, modulo *Skip*s. Because there is no recursion in local protocols, all traces are finite.

The local state of a party σ consists of an assignment of local variables to values and a *mailbox* which stores pending received messages. We do not model the actions of other parties in the system: it is assumed that the environment may nondeterministically deliver messages into the mailbox at any point during execution. We write $\sigma(v)$ for the value of variable v, $\sigma[\nu/v]$ to replace the binding of v with the value ν, and interpret σ as a multiset of messages $m(v = \nu)$ where unambiguous, using the set operations $\cup$ and $\in$.

We define two single-step relations $\langle \sigma, A \rangle \xrightarrow{A} \sigma'$ and $\langle \sigma, L \rangle \xrightarrow{L} \langle \sigma', L', E \rangle$ for local actions and protocols respectively. $\xrightarrow{L}$ relates the reduction of a local protocol to the event it produces, and $\xrightarrow{A}$ is defined separately so that it may be reused below.

$$\boxed{\text{A-SEND}}\quad \frac{}{\langle \sigma, \to r : m(v = \nu)\rangle \xrightarrow{A} \sigma} \qquad \boxed{\text{A-RECV}}\quad \frac{\sigma = m(v = \nu) \cup \sigma'}{\langle \sigma, s \to : m(v)\rangle \xrightarrow{A} \sigma'[\nu_1/v_1]}$$

$$\boxed{\text{A-ASSIGN}}\quad \frac{\langle \sigma, e\rangle \Downarrow \nu}{\langle \sigma, v = e\rangle \xrightarrow{A} \sigma[\nu/v]} \qquad \boxed{\text{L-SKIP}}\quad \frac{}{\langle \sigma, skip\rangle \xrightarrow{L} \langle \sigma, skip, Skip\rangle}$$

$$\boxed{\text{L-SEND}}\quad \frac{\langle \sigma, \to r : m(v = \nu)\rangle \xrightarrow{A} \sigma \qquad \langle \sigma, e\rangle \Downarrow \nu \qquad ...}{\langle \sigma, \to r : m(v = \nu)\rangle \xrightarrow{L} \langle \sigma, skip, Sent\ m(v = \nu)\rangle}$$

$$\boxed{\text{L-RECV}}\quad \frac{\langle \sigma, s \to : m(v)\rangle \xrightarrow{A} \sigma'}{\langle \sigma, s \to : m(v)\rangle \xrightarrow{L} \langle \sigma', skip, Received\ m(v = \nu)\rangle}$$

$$\boxed{\text{L-ASSIGN}}\quad \frac{\langle \sigma, e\rangle \Downarrow \nu}{\langle \sigma, v = e\rangle \xrightarrow{L} \langle \sigma, skip, Assigned\ v = \nu\rangle} \qquad \boxed{\text{L-SEQ}}\quad \frac{\langle \sigma, L_a\rangle \xrightarrow{L} \langle \sigma', L'_a, E\rangle}{\langle \sigma, L_a\ ;\ L_b\rangle \xrightarrow{L} \langle \sigma', L'_a\ ;\ L_b, E\rangle}$$

$$\boxed{\text{L-DISJ}}\quad \frac{\langle \sigma, L_a\rangle \xrightarrow{L} \langle \sigma', L'_a, E\rangle}{\langle \sigma, L_a \vee L_b\rangle \xrightarrow{L} \langle \sigma', L'_a, E\rangle} \qquad \boxed{\text{L-PAR}}\quad \frac{\langle \sigma, L_a\rangle \xrightarrow{L} \langle \sigma', L'_a, E\rangle}{\langle \sigma, L_a \mid\mid L_b\rangle \xrightarrow{L} \langle \sigma', L'_a \mid\mid L_b, E\rangle}$$

$$\boxed{\text{L-FORALL}}\quad \frac{P = \{p_1, p_2, ...\} \qquad \langle \sigma, L[p_1/v]\rangle \xrightarrow{L} \langle \sigma', L', E\rangle}{\langle \sigma, \forall v \in P.\ L\rangle \xrightarrow{L} \langle \sigma', L' \mid\mid L[p_2/v] \mid\mid ..., E\rangle}$$

$$\boxed{\text{L-WHEN-T}}\quad \frac{\langle \sigma, e\rangle \Downarrow true \qquad \langle \sigma, L\rangle \xrightarrow{L} \langle \sigma', L', E\rangle}{\langle \sigma, e \Rightarrow^* L\rangle \xrightarrow{L} \langle \sigma', L', E\rangle} \qquad \boxed{\text{L-WHEN-F}}\quad \frac{\langle \sigma, e\rangle \Downarrow false}{\langle \sigma, e \Rightarrow^* L\rangle \xrightarrow{L} \langle \sigma, e \Rightarrow^* L, Skip\rangle}$$

$\langle \sigma, L\rangle \xrightarrow{L*} \langle \sigma', L', \tau\rangle$ is the reflexive, transitive closure of $\xrightarrow{L}$, relating multiple reduction steps to the resulting trace.

$$\boxed{\text{LS-BASE}}\quad \frac{}{\langle \sigma, L\rangle \xrightarrow{L*} \langle \sigma, L, ()\rangle} \qquad \boxed{\text{LS-IND}}\quad \frac{\langle \sigma_1, L_1\rangle \xrightarrow{L} \langle \sigma_2, L_2, E\rangle \qquad \langle \sigma_2, L_2\rangle \xrightarrow{L*} \langle \sigma_3, L_3, \tau\rangle}{\langle \sigma_1, L_1\rangle \xrightarrow{L*} \langle \sigma_3, L_3, E\,\tau\rangle}$$

Monitors are defined as in Sec. 4.3, and produced from a local protocol by the $\xrightarrow{D}$ function in Sec. 9.7.

Intuitively, monitors are transition systems. Any action in a monitor may execute if its precondition is true, effecting its postcondition.

The semantics of a monitor is given by the relation $\langle \sigma, c, M\rangle \xrightarrow{M} \langle \sigma', c', E\rangle$, relating a monitor to the event it produces upon executing an action.

In addition to local state σ, monitors have *control state* c, which is a mapping of concrete thread identifiers to action ids.

$$\boxed{\text{M-STEP}}\quad \frac{pre_L(\sigma) \qquad pre_C(c) \qquad \langle \sigma, A\rangle \xrightarrow{A} \sigma' \qquad c' = post_C(c)}{\langle \sigma, c, \{\{pre_L\}\{pre_C\}\ A\ \{post_C\}\} \cup M'\rangle \xrightarrow{M} \langle \sigma', c', E\rangle}$$

$$\boxed{\text{M-STEP-EMPTY}}\quad \frac{}{\langle \sigma, c, \{\}\rangle \xrightarrow{M} \langle \sigma, c, \mathit{Skip}\rangle}$$

The reflexive, transitive closure $\langle \sigma, c, M\rangle \xrightarrow{M*} \langle \sigma', c', \tau\rangle$ is defined similarly. Unlike local protocols, monitors do not get 'smaller' as they execute; actions merely become enabled or disabled depending on their preconditions as local and control state changes. Because traces are finite, monitor executions are also finite.

$$\boxed{\text{MS-BASE}}\quad \frac{}{\langle \sigma, c, M\rangle \xrightarrow{M*} \langle \sigma, c, ()\rangle}$$

$$\boxed{\text{MS-IND}}\quad \frac{\langle \sigma_1, c, M\rangle \xrightarrow{M} \langle \sigma_2, c, a\rangle \qquad \langle \sigma_2, c, M\rangle \xrightarrow{M} \langle \sigma_3, c, \tau\rangle}{\langle \sigma_1, c, M\rangle \xrightarrow{M*} \langle \sigma_3, c, a\,\tau\rangle}$$

We also define a function $\mathit{compile}(L) = M$ where $\langle (), \mathit{true}, L\rangle \xrightarrow{D} \langle M, \_\rangle$, as a simpler means of translating a local protocol into a monitor.

The following statements relate monitor and local protocol execution.

Lemma 1 (Monitor Progress). *Given* $compile(L) = M$*, if* $\langle \sigma, L\rangle \xrightarrow{L} \langle \sigma', L', E\rangle$*, then* $\langle \sigma, c, M\rangle \xrightarrow{M} \langle \sigma', c', E\rangle$.

In other words, all monitors compiled from local protocols will always take the same single step or stutter together.

Proof. By induction on the structure of L.

- If $L = skip$, $M = \{\}$. Reduction of L will produce a *Skip*, and by M-STEP-EMPTY, the monitor will also stutter.
- If L is a local action, $M = \{\{LP\}\{t@Start\}\ A\ \{post_C\}\}$. The conclusion follows from M-STEP the fact that the preconditions of the single action in M will always be met: LP must be true because L' exists, and $t@Start$ must be true because it is true initially and there is no other action in the monitor.
- If L is of the form $L_a\,;L_b$, $L_a \vee L_b$, or $L_a \,||\, L_b$, we have $\langle \sigma, L_a\rangle \xrightarrow{L} \langle \sigma', L'_a, E\rangle$ by L-SEQ, L-DISJ, and L-PAR respectively. Immediate from the induction hypothesis.
- If L is of the form $\forall v \in P.\ L_a$, we have $\langle \sigma, L_a\rangle \xrightarrow{L} \langle \sigma', L_a[p_1/v]' \,||\, L_a[p_2/v] \,||\, ..., E\rangle$. Immediate from the induction hypothesis.
- If L is of the form $e \Rightarrow^* L_a$, it may take a step by either L-WHEN-T or L-WHEN-F.

  L-WHEN-T: $\langle \sigma, L_a\rangle \xrightarrow{L} \langle \sigma', L'_a, E\rangle$. Immediate from the induction hypothesis.

  L-WHEN-F: $\langle \sigma, L_a\rangle \xrightarrow{L} \langle \sigma, L_a, Skip\rangle$. By M-STEP-EMPTY, the monitor will also stutter. □

For a monitor to soundly enforce the conformance of an implementation to a local protocol, a *stuck* monitor must always indicate nonconformance. $\xrightarrow{M*}\!\!\!\!\!\!/\ \ $ and $\xrightarrow{L*}\!\!\!\!\!\!/\ \ $ formalize "getting stuck", denoting that the respective relations do not hold.

Theorem 4 (Monitor Synthesis Soundness). *Given* $compile(L) = M$*, if* $\langle \sigma, c, M\rangle \xrightarrow{M*}\!\!\!\!\!\!/\ \ $*, then* $\langle \sigma, L\rangle \xrightarrow{L*}\!\!\!\!\!\!/\ \ $.

We prove the contrapositive: if $\langle \sigma, L\rangle \xrightarrow{L*} \langle \sigma', L', \tau\rangle$, then $\langle \sigma, c, M\rangle \xrightarrow{M*} \langle \sigma', c, \tau\rangle$. In other words, for all traces, if the local protocol does not get stuck, then the monitor does not get stuck.

Proof. By induction on a derivation of $\xrightarrow{L*}$.

Base case ($\tau = ()$): we must prove $\langle \sigma, c, M\rangle \xrightarrow{M*} \langle \sigma, c, ()\rangle$. True by MS-BASE.

Inductive case ($\tau = E\,\tau'$): we must prove $\langle \sigma, c, M\rangle \xrightarrow{M*} \langle \sigma', c', E\,\tau'\rangle$, given induction hypotheses

$$\langle \sigma, L\rangle \xrightarrow{L} \langle \sigma'', L'', E\rangle \tag{IH1}$$

$$\langle \sigma'', L'' \rangle \xrightarrow{L*} \langle \sigma', L', \tau' \rangle \tag{IH2}$$

By MS-IND, it suffices to instead prove

$$\langle \sigma, c, M \rangle \xrightarrow{M} \langle \sigma'', c'', E \rangle \tag{G1}$$

$$\langle \sigma'', c'', M \rangle \xrightarrow{M*} \langle \sigma', c', \tau' \rangle \tag{G2}$$

By induction on the structure of L,

- If $L = skip$, $compile(L) = M = \{\}$. The conclusion is immediate from Lemma 1 and MS-BASE.
- If L is of the form $\rightarrow r : m(v_1 = \nu_1, ...)$ and $thread(L, t)$, $compile(L) = M = \{\{pre_L\}\{t@Start\} \rightarrow r : m(v_1 = \nu_1, ...)) \ \{t@id\}\}$. (G1) is immediate from Lemma 1. By (IH1), (IH2), and L-SEND, we know that $L' = skip$ and $\tau' = ()$. (G2) follows from MS-BASE.
- The proofs for the other local actions $s \rightarrow: m(v)$ and $v = e$ are very similar, except for using L-RECV and L-ASSIGN, respectively.
- If L is of the form $L_a \,;\, L_b$ where $compile(L_a) = M_a, compile(L_b) = M_b$, we have induction hypotheses of the form

$$\langle \sigma, L_a \rangle \xrightarrow{L*} \langle \sigma', L'_a, \tau_a \rangle \implies \langle \sigma, c, M_a \rangle \xrightarrow{M*} \langle \sigma', c', \tau_a \rangle \tag{IH3}$$

$$\langle \sigma, L_b \rangle \xrightarrow{L*} \langle \sigma', L'_b, \tau_b \rangle \implies \langle \sigma, c, M_b \rangle \xrightarrow{M*} \langle \sigma', c', \tau_b \rangle \tag{IH4}$$

  From (IH1), L can take a step by L-SEQ. We have $\langle \sigma, L_a \rangle \xrightarrow{L} \langle \sigma', L'_a, E \rangle$.
  From Lemma 1, we have $\langle \sigma, c, M_a \rangle \xrightarrow{M} \langle \sigma', c', E \rangle$, giving us (G1).
  We also know $\langle \sigma', L'_a \,;\, L_b \rangle \xrightarrow{L*} \langle \sigma'', L', \tau' \rangle$ by (IH2).
  By L-SEQ, LS-IND, and (IH3), $\langle \sigma', c', M_a \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_a \rangle$.
  By L-SEQ, LS-IND, and (IH4), $\langle \sigma'', c'', M_b \rangle \xrightarrow{M*} \langle \sigma''', c''', \tau_b \rangle$.
  From the definition of $\xrightarrow{D}$, all actions in M_b have a precondition which is transitively met only after some action in M_a has executed. This means that M_b's actions are entirely sequenced after M_a's.
  Hence $\tau = \tau_a \tau_b$ and $\langle \sigma', c', M_a \cup M_b \rangle \xrightarrow{M*} \langle \sigma''', c''', \tau_a \tau_b \rangle$, giving us (G2).
- If L is of the form $L_a \,||\, L_b$, we have induction hypotheses of the form

$$\langle \sigma, L_a \rangle \xrightarrow{L*} \langle \sigma', L'_a, \tau_a \rangle \implies \langle \sigma, c, M_a \rangle \xrightarrow{M*} \langle \sigma', c', \tau_a \rangle \tag{IH3}$$

$$\langle \sigma, L_b \rangle \xrightarrow{L*} \langle \sigma', L'_b, \tau_b \rangle \implies \langle \sigma, c, M_b \rangle \xrightarrow{M*} \langle \sigma', c', \tau_b \rangle \tag{IH4}$$

  From (IH1), L can take a step by L-PAR. We have $\langle \sigma, L_a \rangle \xrightarrow{L} \langle \sigma', L'_a, E \rangle$.
  From Lemma 1, we have $\langle \sigma, c, M_a \rangle \xrightarrow{M} \langle \sigma', c', E \rangle$, giving us (G1).
  We also know $\langle \sigma', L'_a \,||\, L_b \rangle \xrightarrow{L*} \langle \sigma'', L', \tau' \rangle$ by (IH2).

By L-PAR, LS-IND, and (IH3), $\langle \sigma', c', M_a \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_a \rangle$.
By L-PAR, LS-IND, and (IH4), $\langle \sigma', c', M_b \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_b \rangle$.
In other words, both M_a and M_b may independently take multiple steps forward.
From the definition of $\xrightarrow{D}$ and *thread*, the set of threads in L_a and L_b (and hence the *domains* of the control preconditions of M_a and M_b) are disjoint. This means that M_a's actions may interleave arbitrarily with M_b's, and they cannot invalidate each other's control preconditions and cause either monitor to get stuck that way.[3]
Hence $\langle \sigma', c', M_a \cup M_b \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_i \rangle$, where τ_i is some interleaving of τ_a and τ_b, giving us (G2).

– If L is of the form $\forall v \in P.\ L_a$, we have induction hypotheses of the form

$$\forall p \in P.\ \langle \sigma, L_p \rangle \xrightarrow{L*} \langle \sigma', L'_p, \tau_p \rangle \implies \langle \sigma, c, M_a \rangle \xrightarrow{M*} \langle \sigma', c, \tau_p \rangle \qquad \text{(IH3)}$$

where $L_p = L_a[p/v]$ and $M_a = \mathit{compile}(L_a)$. Notably, while there are conceptually $|P|$ threads, this value is not statically known, so there is no sub-monitor for each L_p; instead, the relevant actions of M_a which capture the semantics of L_a are parameterized by p.
From (IH1), L can take a step by L-FORALL. We have $\langle \sigma, L_a \rangle \xrightarrow{L} \langle \sigma', L_a[p_1/v]' || L_a[p_2/v] \,||\, ..., E \rangle$.
From Lemma 1, we have $\langle \sigma, c, M_a \rangle \xrightarrow{M} \langle \sigma', c', E \rangle$, giving us (G1).
We also know $\langle \sigma', L_a[p_1/v]' \,||\, L_a[p_2/v] \,||\, ... \rangle \xrightarrow{L*} \langle \sigma'', L', \tau' \rangle$ by (IH2).
By L-PAR, LS-IND, and (IH3), $\langle \sigma', c', M_a \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_a \rangle$.
In other words, each parameterized monitor M_p may independently take multiple steps forward. The reasoning is the same as for $L_a \,||\, L_b$, but with an unknown number of threads.
Hence $\langle \sigma', c', M_a \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_a \rangle$, giving us (G2).

– If L is of the form $L_a \vee L_b$, we have induction hypotheses of the form

$$\langle \sigma, L_a \rangle \xrightarrow{L*} \langle \sigma', L'_a, \tau_a \rangle \implies \langle \sigma, c, M_a \rangle \xrightarrow{M*} \langle \sigma', c', \tau_a \rangle \qquad \text{(IH3)}$$

$$\langle \sigma, L_b \rangle \xrightarrow{L*} \langle \sigma', L'_b, \tau_b \rangle \implies \langle \sigma, c, M_b \rangle \xrightarrow{M*} \langle \sigma', c', \tau_b \rangle \qquad \text{(IH4)}$$

From (IH1), L can take a step by L-DISJ. We have $\langle \sigma, L_a \rangle \xrightarrow{L} \langle \sigma', L'_a, E \rangle$.
From Lemma 1, we have $\langle \sigma, c, M_a \rangle \xrightarrow{M} \langle \sigma', c', E \rangle$, giving us (G1).
We also know $\langle \sigma', L'_a \rangle \xrightarrow{L*} \langle \sigma'', L', \tau' \rangle$ by (IH2).
By L-DISJ, LS-IND, and (IH3), $\langle \sigma', c', M_a \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_a \rangle$.
In other words, only M_a (or symmetrically, only M_b) continues to take multiple steps forward.

[3] Invalidation of *logical* preconditions may still occur, but this is a concern orthogonal to parallel composition.

By the definition of $\xrightarrow{D}$ and *thread*, M_1's and M_2's control preconditions refer to the same thread. This is the crucial difference between $\vee$ and $||$, and the reason why only one monitor's actions may execute: it invalidates the other's control precondition.

Hence $\langle \sigma', c', M_a \cup M_b \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_a \rangle$, giving us (G2).

– If L is of the form $e \Rightarrow^* L_a$, we have induction hypotheses of the form

$$\langle \sigma, L_a \rangle \xrightarrow{L*} \langle \sigma', L'_a, \tau_a \rangle \implies \langle \sigma, c, M_a \rangle \xrightarrow{M*} \langle \sigma', c', \tau_a \rangle \qquad \text{(IH3)}$$

From (IH1), L can take a step by L-WHEN-T or L-WHEN-F.

L-WHEN-T: we have $\langle \sigma, L_a \rangle \xrightarrow{L} \langle \sigma', L'_a, E \rangle$.

From Lemma 1, we have $\langle \sigma, c, M_a \rangle \xrightarrow{M} \langle \sigma', c', E \rangle$, giving us (G1).

We also know $\langle \sigma', L'_a \rangle \xrightarrow{L*} \langle \sigma'', L', \tau' \rangle$ by (IH2).

Together with (IH3), $\langle \sigma', c', M_a \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_a \rangle$, giving us (G2).

L-WHEN-F: we have $\langle \sigma, L_a \rangle \xrightarrow{L} \langle \sigma, L_a, \mathit{Skip} \rangle$.

Therefore, $\langle \sigma, c, M_a \rangle \xrightarrow{M} \langle \sigma, c, \mathit{Skip} \rangle$, giving us (G1).

We also know $\langle \sigma, L'_a \rangle \xrightarrow{L*} \langle \sigma', L', \tau' \rangle$ by (IH2). By (IH3), $\langle \sigma', c', M_a \rangle \xrightarrow{M*} \langle \sigma'', c'', \tau_a \rangle$, giving us (G2). □

9.9 Single-decree Paxos

See Fig. 14, which renders the protocol specification and original natural-language specification [30] side-by-side to demonstrate the direct correspondence.

```
// Initialization omitted

forall p in P
  (p.proposal = p.proposal + 1;
  forall a in A
    p->a: prepare(n=p2i(p) *
      100 + p.proposal);

    n > a.highest_proposal =>
      a.highest_proposal = n;
      a->p: promise(
        cp=a.accepted_proposal,
        cv=a.accepted_value);
      p.resp =
        union(p.resp, {a});
      cp > 0 & cp > p.cp =>
        p.cp = cp;
        p.value = cv

  ||

  size(p.resp) > p.majority =>*
    forall a1 in p.resp
      p->a1: propose(
        pn=p.proposal, pv=p.value);



      ac2 = a1;
      pn == a1.highest_proposal =>
        a1.accepted_proposal = pn;
        a1.accepted_value = pv;
        a1->p: accept;


        forall l in L
          ac2->l: accept)
```

Putting the actions of the proposer and acceptor together, we see that the algorithm operates in the following two phases.
Phase 1. (a) A proposer selects a proposal number n and sends a prepare request with number n to a majority of acceptors.

(b) If an acceptor receives a prepare request with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

Phase 2. (a) If the proposer receives a response to its prepare requests (numbered n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

To learn that a value has been chosen, a learner must find out that a proposal has been accepted by a majority of acceptors. The obvious algorithm is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the proposal.

Fig. 14: Single-decree Paxos description and specification