Protocol Conformance with Choreographic PlusCal

Darius Foo[0000−0002−3279−5827], Andreea Costea[0000−0002−9089−9392], and Wei-Ngan Chin[0000−0002−9660−5682]

National University of Singapore
{dariusf,andreeac,chinwn}@comp.nus.edu.sg

Abstract. Distributed protocols, an essential part of modern computing infrastructure, are well-known to be difficult to implement correctly. While lightweight formal methods such as TLA$^+$ can be effectively used to verify abstract protocols, end-to-end validation of real-world protocol implementations remains challenging due to their complexity. To address this problem, we extend the TLA$^+$ toolset along two fronts. We propose several extensions to PlusCal – an algorithm language which compiles to TLA$^+$ – to allow writing distributed protocols as choreographies. This enables more structured and succinct specifications for role-based protocols. We also provide a methodology and toolchain for compiling TLA$^+$ models into monitors, allowing them to be used to test existing systems for conformance. The result is a lightweight testing method that bridges specification and implementation. We demonstrate its benefits with case studies of both classic and recent protocols and show it to be readily applicable to existing systems with low runtime overhead.

1 Introduction

Distributed systems are an essential part of modern computing infrastructure. They are well known to be hard to implement correctly: Faults and asynchrony give rise to huge state spaces which make ad hoc testing ineffective. To alleviate this, developers of distributed systems implement well-known distributed protocols and algorithms, e.g. Paxos and Raft, which provide strong guarantees, e.g. solving consensus provided a quorum of nodes does not fail. Implementing a protocol correctly transfers its guarantees to the implementation, but this is easier said than done. In practice, production-ready consensus implementations are large and complex, containing optimizations and low-level details that make the correspondence with an abstract protocol difficult to establish. Moreover, protocols must often be extended to meet real-world needs. It is thus difficult to say if a protocol is correct, and if a system indeed implements a given protocol.

Developers working with such protocols today rely on *lightweight formal methods* [28, 6], best exemplified by TLA$^+$. Its combination of first-order temporal logic with explicit state-space model checking is simple, yet effective. Its expressiveness makes it readily applicable, and its intended workflow of checking protocol properties in a relatively low-commitment manner is compatible with software engineering constraints [28]. However, TLA$^+$ has two shortcomings.

Distributed protocols are specified in TLA$^+$ as state machines, consisting of a set of *actions* (relations between states). This form is simple and explicit, but relatively unstructured. Control flow must be encoded in guard conditions, and it is generally difficult to see how actions relate: given two actions A and B, we must examine potentially all other actions to see that B follows A and is the *only* action that could follow A. Modifying protocols is in turn error-prone, as seemingly local protocol changes often require nonlocal specification changes. Even for experienced users, changing a large model in state machine form without introducing bugs takes several tries. TLA$^+$ has an algorithm language, PlusCal [23], that should solve these issues, but despite being more structured and apt, it is not often used[1]. We suggest that this is because it does not provide sufficiently high-level constructs for common patterns that occur in distributed protocols.

The TLA$^+$ toolset also does not contain a lightweight mechanism for checking an implementation's conformance to a specification. This is a pity given the ready availability of protocol specifications and the abundance of supposedly-conforming implementations. We thus extend TLA$^+$ along two fronts.

We first extend PlusCal with *choreographies* – global descriptions of communication protocols, and a natural fit for specifying distributed behavior. Choreographies are typically used for specifying communication protocols using multiparty session types [15], in security protocols ("Alice-and-Bob notation"), or writing distributed programs. As specifications, they read causally and resemble informal natural-language descriptions. Choreographies are *projected* into individual `process` declarations and subsequently translated to regular PlusCal. The translation is parametric in the choice of network semantics.

Secondly, we extend the TLA$^+$ tools with a source-to-source translator from TLA$^+$ into a *monitor* in an implementation language. This allows a system model to be linked into an implementation and executed in lockstep to check for protocol conformance, a lightweight testing method which does not require invasive changes. Given a TLA$^+$ model verified using the TLC model checker, an implementation which refines it thus inherits its safety properties.

We implement both of these extensions in the TLA$^+$ tools and demonstrate both the use of choreographies and monitors with case studies.

Our prototype, experimental data and results, and an extended technical report containing case studies and more detailed proofs are publicly available[2].

2 Overview

We highlight our proposed extensions and methodology using an instance of the classic two-phase commit protocol [24], in which a set of database replicas collectively agree on whether or not to commit a distributed transaction.

The two-phase commit 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 decide independently

[1] Of the 99 TLA$^+$ models in the official examples repository, 61 could be called distributed protocols and only 15 of them use PlusCal.

[2] `https://github.com/dariusf/tlaplus/tree/cpcal`

```
process (C \in coordinators)
  variables temp = participants,
            aborted = FALSE; {
    while (temp /= {}) {
      with (r \in temp) {
        Send(self, r, "prepare");
        temp := temp \ {r};
      } };
    temp := participants;
    while (temp /= {} \/ aborted) {
      with (r \in temp) {
        either {
        Receive(r, self, "prepared");
        } or {
        Receive(r, self, "abort");
        aborted := TRUE;
        };
        temp := temp \ {r};
      } };
    if (aborted) {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "abort");
        temp := temp \ {r};
      } };
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "aborted");
        temp := temp \ {r};
      } }
    } else {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "commit");
        temp := temp \ {r};
      } }
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "committed");
        temp := temp \ {r};
      } } } }
```

Fig. 1: Two-Phase Commit Coordinator (PlusCal)

```
process (P \in participants) {
  Receive(coord, self, "prepare");
  either {
  psend:
    Send(self, coord, "prepared");
  } or {
    Send(self, coord, "abort");
  };
  either {
    Receive(coord, self, "commit");
    Send(self, coord, "committed");
  } or {
    Receive(coord, self, "abort");
    Send(self, coord, "aborted");
  } }
```

Fig. 2: Two-Phase Commit Participant (PlusCal)

```
choreography
  (P \in participants),
  (C \in coordinators) {
  task coordinators "phase1" {
    all (p \in participants) {
      Transmit(coord, p, "prepare");
      either {
        Transmit(p, coord, "prepared");
      } or {
        Transmit(p, coord, "aborted");
        cancel "phase1";
      } } };
  if (aborted) {
    all (p \in participants) {
      Transmit(coord, p, "abort");
      Transmit(p, coord, "aborted");
    }
  } else {
    all (p \in participants) {
      Transmit(coord, p, "commit");
      Transmit(p, coord, "committed");
    } } }
```

Fig. 3: Two-Phase Commit Choreography

whether or not to commit and reply to the coordinator, which decides to commit the transaction provided *all* participants decided to commit. Entering the second phase of the protocol, the coordinator performs another multicast to inform the participants of the outcome, which end the protocol by acknowledging.

The protocol in PlusCal is shown in Fig. 1 and Fig. 2. It is parameterized by two *roles* (Def. 1), `participants` and `coordinators`: finite sets of identifiers (or *parties*) which represent protocol nodes. Intuitively, parties in a role enact the same unique pattern of interactions in a protocol. One PlusCal `process` per role is declared, and processes are replicated for each party in their role. `Send` and `Receive` are macros for communication. `P` and `C` are process names (Sec. 3.1).

Fig. 1 shows the coordinator's process; `coordinators` is the singleton role `{coord}`. Within its body, `self` refers to the current party (which is always `coord` here). There are two locals: a set of participants `temp` and a boolean variable `aborted`. The initial multicast is modelled with a `while` loop which repeatedly selects a participant to send a `prepare` message to by iterating over `temp`.

The loop may seem a rather low-level construct to use, but while it is possible to define a TLA$^+$ operator to perform an atomic multicast and send a batch of messages at once, the loop lets us model possible reorderings of parallel sends. Unfortunately, PlusCal macros may only be parameterized by expressions, not blocks or statements, so this pattern of relying on loops and `temp` to communicate with the participants is repeated throughout the coordinator's model.

The second loop in the coordinator's process receives `prepared` messages from every participant, and stops at the first `abort`, ending the first phase. The second phase begins with the coordinator making a decision: if a participant aborted the transaction, the coordinator ensures that every participant acknowledges this; otherwise it solicits `committed` messages from everyone.

Moving on to the participant model in Fig. 2, we see that it mirrors the coordinator: the control flow is identical, sends are replaced with receives, loops with replicated processes, and `if` with `either`.

2.1 Choreographies

It may have seemed natural to break the specification of the two-phase commit protocol into two parts corresponding to the coordinator and participant, respectively. However, such a split compromises the readability and maintainability of the protocol: it is tedious to identify the matching receives and sends spread across different `process`es and keep them in sync. A better specification approach is a *choreography*, a global description of interactions. The entire two-phase commit protocol as a choreography is shown in Fig. 3. Like a program, a choreography has imperative and control flow constructs, but is written from a global perspective instead: statements like `Transmit` are now allowed to mention members of *multiple* roles. `Transmit` is intuitively a juxtaposed `Send` on one party and a `Receive` on another, and effectively halves the size of protocol specifications. A `choreography` is statically *projected* (Sec. 4) into multiple `process`es, recovering something close to what was written in Fig. 1 and Fig. 2.

2.2 Parallel Composition and `all`

There are still a number of problems with the PlusCal model. A glaring one is that the coordinator's model is overly sequential: despite our attempt to model parallelism with the `while` loop in Fig. 1, the coordinator is forced to send all its *prepare* messages before receiving a single response. To alleviate that, we could merge the first two loops, but that makes message transmission synchronous. To specify asynchronous transmissions in PlusCal, we could factor the sends and receives into separate `process`es and manually add variables and `await`s to sequence them between phases, but this is, again, error-prone and tedious.

At this point one might be tempted to fall back to TLA$^+$, where having separate actions for sending and receiving messages is easy. The downside is that TLA$^+$ is *unstructured*. Any action may interact with any other, so it is difficult to see where new actions fit into an existing model without a global view. Furthermore, TLA$^+$ is no less verbose; the PlusCal translator's TLA$^+$ output is more than twice the size of Fig. 1 and Fig. 2 combined.

To overcome this, we extend PlusCal with two structuring mechanisms for intra-process concurrency, `all` and `par`; use of the former is illustrated in Fig. 3. The `all` statement may be thought of procedurally as a *fork-join* or nested `process`, and logically as universal quantification. Its body is executed once for each element in the given set, *possibly in parallel*.

`all` in combination with `Transmit` is particularly useful for modelling multicasts. The resulting model is more succinct, but more importantly, it is more *abstract*: the body of the `all` statement *need not* execute sequentially. Sends and receives to different participants may also occur asynchronously.

Parallel composition `par { ... } and { ... } ...` is similar: each of the given blocks executes in parallel and may differ, compared to what `all` does, where each element of the set gives rise to a copy of a single process.

2.3 Tasks and Cancellation

A second problem with the PlusCal model in Fig. 1 is that *cancellation*, a useful complement to timeouts, is done in an ad hoc manner. While we attempted an early `abort` in the first phase, we had to send all the `prepare` messages first.

To address this, we extend PlusCal with two constructs, `task` and `cancel`. `task` delimits a block of statements; anything within will *no longer execute* following the corresponding cancellation. `cancel` terminates its matching `task`.

The model finally expresses the desired behavior. Suppose we observe (a prefix of) a behavior where two `prepare` messages are sent, immediately after which a reply to `abort` comes back. In the final model, no more `prepare` messages will be sent, and no more `prepared` or `abort` messages will be *received*. In other words, the entire first phase has been cancelled early, and the decision in the second phase will be the very next thing following the cancellation.

With all these changes implemented, we see also that our choreographic specification is very succinct: the final model in Fig. 3 is less than half the size of its PlusCal counterpart while saying more.

```
psend(self) ==
  /\ pc[self] = "psend"
  /\ outbox' = [outbox EXCEPT ![self] =
    outbox[self] \union
      {[To |-> coord, From |-> self,
        Type |-> "prepared"]}]
  /\ pc' = [pc EXCEPT ![self] = "Lbl_2"]
  /\ UNCHANGED<< participants, temp,
                 aborted >>
```

(a) TLA$^+$ rendering of psend action

```
psend(self) ==
  /\ pc = "psend"
  /\ outbox' = outbox \union
    {[To |-> coord, From |-> self,
      Type |-> "prepared"]}
  /\ pc' = "Lbl_2"
```

(b) Projection of psend action

Fig. 5: TLA$^+$ rendering and projection of psend action from Fig. 2

2.4 Verification and Monitoring

Choreographic PlusCal projects and translates down to regular PlusCal, which in turn translates to TLA$^+$, so it fits naturally into the verification pipeline of the TLA$^+$ tools (Fig. 4).

Suppose we specify an invariant that the protocol must uphold and verify it with TLC, the TLA$^+$ model checker. For example, an invariant for two-phase commit is that participants all commit or all abort, i.e. we cannot observe one participant that has committed and another that has aborted.

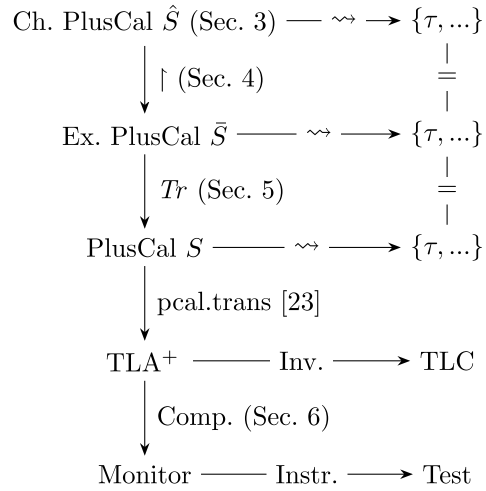


Fig. 4: Overview

We would like assurance that a given implementation of two-phase commit follows the protocol and satisfies the same invariant. The typical way to ensure this is to show that the implementation is a refinement of the model: for each of its behaviors, we find a corresponding behavior of the model that "justifies" the implementation behavior under a refinement mapping— a mapping of implementation states to model states. This in turn implies inheritance of safety properties.

We provide a lightweight way to gain this assurance for TLA$^+$ models via monitoring. As mentioned earlier, this supports the validation of the many hand-tuned implementations which already exist, not requiring major changes.

We illustrate monitor generation for the statement labelled psend in Fig. 2, where a participant sends a message to the coordinator, is shown in Fig. 5a. This translates down to an action which adds to a participant-local outbox, then advances a program counter pc to capture the protocol's progression.

Given a TLA$^+$ model satisfying some syntactic conditions[3] (e.g. that local variables are represented as records), we project it, removing non-local variables and actions and projecting local record variables. psend now looks as in Fig. 5b.

[3] Choreographic PlusCal ensures these by construction, but they can also be satisfied in handwritten TLA$^+$ models, so this methodology applies to them equally.

We then generate code to be linked with an implementation to enable the collection and validation of traces. Collection is done via the following function.

```
func (m *Monitor) Capture(s State, a Action, args ...TLA)
```

The `State` struct is generated from the variables of the model and represents its abstract state via a deep embedding of TLA$^+$. The user provides a refinement mapping by constructing it using the concrete state of the implementation, and placing it at appropriate *linearization points* to snapshot the state of the system.

With traces captured, validation is done by interpreting projected actions as assertions and using them to check traces, via generated code like in Fig. 6.

```go
func psend(prev state, this state, self TLA) bool {
  if !(reflect.DeepEqual(prev.pc, Str("psend"))) {
    return false
  }
  // ... outbox check elided
  if !(reflect.DeepEqual(this.pc, Str("Lbl_2"))) {
    return false
  }
  return true
}
```

Fig. 6: Go rendering of `psend` in generated monitor

This gives us a pipeline for checking the correspondence of a high-level choreography with a lower-level implementation, supporting their evolution in tandem. As this is a testing approach, it finds only *violations* of refinement, but scales to real-world implementations.

In the following sections we describe a core fragment of PlusCal and define our extensions to it (Sec. 3). Sec. 4 and Sec. 5 define projection and translation, respectively, to this core fragment. Sec. 6 describes the generation and use of monitors. We evaluate the approach empirically in Sec. 7 before concluding.

3 Choreographic PlusCal

3.1 PlusCal

We begin by introducing the syntax (Fig. 7a) and semantics of a minimal subset of PlusCal, *Core PlusCal*, which extends TLA$^+$ with `process` declarations and control flow constructs such as conditionals and loops.

A model *pcal* is a sequence of global variable declarations, followed by one or more `process`es, sequential and replicated imperative programs. Processes are declared with an identifying expression of the form $v \in e$, where v is a name for the set of replicas collectively, and e is a set of indices for the individual replicas.

`skip` is a no-op. `await` blocks until a condition becomes true. `with` nondeterministically selects an element of the set e and binds it to v in its body, and `either` nondeterministically executes one of its branches. The rest are standard.

The semantics of a (Core) PlusCal model is a set of *behaviors*, sequences of *states*. A state is an assignment of variables to values; in each state global variables are assigned values, and each local is assigned one value *per process*. The execution of a process gives rise to a behavior by updating variables via assignments, nondeterminism allows several possible behaviors, and the execution of multiple processes interleaves their individual behaviors. Within process bodies, the special variable *self* refers to the current replica.

$$
\begin{aligned}
&\mathit{pcal} ::= \mathit{vardecl}^*\,\mathit{proc}^* \qquad \mathit{vardecl} ::= v = e \qquad \mathit{procid} ::= v \in \{e^*\}\\
&\mathit{proc} ::= \texttt{process}\ \mathit{procid}\ \mathit{vardecl}^*\ S\\
&S ::= \texttt{skip} \mid \texttt{await}\ e \mid \texttt{while}\ e\ S \mid \texttt{if}\ e\ S\ S \mid \texttt{with}\ v \in e\ S \mid v := e\\
&\qquad \mid S;\ S \mid \texttt{either}\ S\ \texttt{or}\ S\\
&e ::= \mathit{string} \mid v \mid e = e \mid e \neq e \mid \forall v \in e : e \mid \{e^*\} \mid e \setminus e \mid e \cup e\\
&\qquad \mid \langle\langle e^* \rangle\rangle \mid e[e] \mid e[e] := e
\end{aligned}
$$

(a) Core PlusCal

$$
\begin{aligned}
\bar{S} ::= &\ S \mid \texttt{Send}(e, e, e)\\
&\mid v := \texttt{Receive}(e, e)\\
&\mid \texttt{all}\ v \in e\ \bar{S} \mid \texttt{par}\ \bar{S}\ \texttt{and}\ \bar{S}\\
&\mid \texttt{task}\ R\ e\ \bar{S} \mid \texttt{cancel}\ e
\end{aligned}
$$

(b) Extended Core PlusCal

$$
\begin{aligned}
&\mathit{cpcal} ::= \mathit{vardecl}^*\ \mathit{chor}\ \mathit{proc}^*\\
&\mathit{chor} ::= \texttt{choreography}\ (\mathit{procid}\ \mathit{vardecl}^*)^*\ \hat{S}\\
&\hat{S} ::= \bar{S} \mid \texttt{while}\ e^*\ \hat{S}\\
&\qquad \mid \texttt{Transmit}(e, e, v = e)
\end{aligned}
$$

(c) Choreographic PlusCal

Fig. 7: Syntax of Core PlusCal and its extensions

3.2 Extensions

We extend Core PlusCal with new statements (Fig. 7b), and a `choreography` top-level declaration in *cpcal* (Fig. 7c), which allows additional global statements.

A protocol implementation typically involves multiple sets of physical *nodes* (or replicas), where nodes within each set follow the same "pattern of interaction". We represent nodes by their identifiers (concretely, TLA$^+$ model values or short strings), and their interactions by PlusCal `process`es. These concepts are made explicit in all our PlusCal variants.

Definition 1 (Roles and parties). *A* role *R is a statically-declared set of node identifiers. A node identifier, as a member of a* role*, is called a* party*.*

The syntax of Extended Core PlusCal also contains explicit *send* and *receive* operations that parties use to exchange messages over the network [32, 39].

`with` executes its body with a nondeterministically chosen element of some set, whereas `all` does so with *every* element of the set. There is no obligation that this be done sequentially, so each "subprocess" executes in parallel. Parallel composition is similar to `all`, but has a *fixed* number of child processes with *different* behavior. Unlike subprocesses in Distributed PlusCal [1], the translation scheme for both is compositional, allowing arbitrary nested trees of processes.

Finally, there are the paired constructs of *tasks* and *cancellation*. A task is a block of statements that may or may not execute to completion. Tasks are local to a given role R, and are uniquely named with an expression e. Correspondingly, cancellation is a local operation which terminates its task: the suffix of any behavior following a cancellation will not contain actions in the scope of its task. There is exactly one task for a given cancellation. A task may, however, be cancelled in multiple places. For example, in

$$\texttt{par}\ (\texttt{cancel}\ e)\ \texttt{and}\ (\texttt{task}\ \{p_1\}\ e\ \ \texttt{Transmit}(p_2, p_2, v = m))$$

if `Transmit` means that p_1 sends the message m asynchronously to p_2 and the network is reliable, then there are two outcomes: one where m is received by p_2, and one where it is sent but never received. Tasks and cancellation are thus useful for more fine-grained modeling of timeouts and aborted operations.

A `choreography` declaration *chor* (Fig. 7c) gives the roles involved in the protocol and their locals. `Transmit` allows for a name `v` to be given, which `e` will be bound to on the receiving party. A `while` loop must have a termination condition for each role its body involves, and is otherwise standard.

4 Projection

The first step in the translation to PlusCal is to split choreographies $\hat{S}$ into processes $\bar{S}$ via projection. As mentioned earlier, a `choreography` declares a number of roles, each with its own local variables. The body of the `choreography` $\hat{S}$ (Fig. 7c) allows two additional *global* statements (`Transmit` and global `while`) and may mention variables and members of all roles simultaneously in expressions, subject to well-formedness conditions detailed in this section.

A global `while` statement has a termination condition for every role involved in its body. We have seen `Transmit` before; a multicast is expressed as a `Transmit` inside `all`. Consider how the first multicast from Fig. 3 is projected:

$$\textit{coordinator}: \ \texttt{all}\ p \in \textit{participants}\ (\texttt{Send}(\textit{coord}, p, \textit{prepare}))$$
$$\textit{participants}: \ v := \texttt{Receive}(p, \textit{coord})$$

The `Transmit` is split into a `Send` and `Receive`. As the set of participants is “quantified over”, the `all` block vanishes when we view the protocol from the perspective of some p, which performs only a single `Receive` from *coord*.

Projection is a partial operation, requiring that variables and expressions used by non-global statements belong to only one role. For example, we cannot `await` a global condition involving multiple roles.

Choreographies are parametric in their choice of network semantics. The user can implement any network semantics they wish via the macros `Send` and `Receive`: reliable, ordered, etc. This is a more minimalistic solution than in Distributed PlusCal [1], which provides explicit `broadcast` and `multicast` operations and multiple kinds of channels, and Modular PlusCal [43, 8], which adds constructs for separating the system from the environment of a model.

Definition 2 (Projection). *The projection of $\hat{S}$ onto a role R, $\hat{S} \upharpoonright R$, is defined inductively over the syntax of $\hat{S}$ in Fig. 8.*

Parties and roles are first-class citizens in Choreographic PlusCal, yet projection is done at translation time and must use only statically-known information. A number of auxiliary predicates which relate syntax with (approximations of) semantic information are used in its definition. $loc(v, R)$ is true if v is a local variable of R, $loc(e, R)$ is true if all variables in e are local to R; $role(e, R)$ and $party(e, R)$ allow interpreting expressions as roles and parties of the given role respectively; $task(e, R)$ holds if there is a task `task` $R\ e\ S$.

Projection is defined in Fig. 8. Assignments, `await` (elided), and `cancel` vanish if the variables, expressions, and/or tasks they mention are not local to the *current role* (the role being projected onto). Transmissions are projected depending on *how* the parties involved are related to the current role. Projection is “pushed through” sequential composition, `par`, and `either`.

$$(v := e) \upharpoonright R \triangleq \begin{cases} v := e \text{ if } \mathit{loc}(v,R), \mathit{loc}(e,R) \\ \texttt{skip} \quad \text{otherwise} \end{cases} \quad (\texttt{cancel}\ e) \upharpoonright R \triangleq \begin{cases} \texttt{cancel}\ e \text{ if } \mathit{task}(e,R) \\ \texttt{skip} \quad \text{otherwise} \end{cases}$$

$$\underset{\mathit{party}(s,R_s),\mathit{party}(r,R_r),R_s \neq R_r}{\texttt{Transmit}(s,r,v=m) \upharpoonright R} \triangleq \begin{cases} \texttt{Send}(s,r,m) & \text{if } \mathit{party}(r,R) \\ v := \texttt{Receive}(s,r) & \text{if } \mathit{party}(s,R) \\ \texttt{skip} & \text{otherwise} \end{cases}$$

$$(\texttt{task}\ Q\ e\ \hat{S}) \upharpoonright R \triangleq \begin{cases} \texttt{task}\ Q\ e\ (\hat{S} \upharpoonright R) & \text{if } Q = R \\ \hat{S} \upharpoonright R & \text{otherwise} \end{cases}$$

$$(\texttt{all}\ v \in e\ \hat{S}) \upharpoonright R \triangleq \begin{cases} \texttt{all}\ v \in e\ (\hat{S} \upharpoonright R) & \text{if } \mathit{role}(e,Q), Q \neq R \\ \texttt{all}\ v \in e\ (\hat{S} \upharpoonright R) & \text{if } \mathit{loc}(e,R) \\ \hat{S} \upharpoonright R & \text{otherwise} \end{cases}$$

$$(\texttt{if}\ e\ \hat{S}_1\ \hat{S}_2) \upharpoonright R \triangleq \begin{cases} \texttt{if}\ e\ (\hat{S}_1 \upharpoonright R)\ (\hat{S}_2 \upharpoonright R) & \text{if } \mathit{loc}(e,R) \\ \texttt{either}\ (\hat{S}_1 \upharpoonright R)\ \texttt{or}\ (\hat{S}_2 \upharpoonright R) & \text{otherwise} \end{cases}$$

$$(\texttt{while}\ e^*\ \hat{S}) \upharpoonright R \triangleq \begin{cases} \texttt{while}\ e_1\ (\hat{S} \upharpoonright R) & \text{if } \mathit{loc}(e_1,R), e_1 \in e^* \\ \hat{S} \upharpoonright R & \text{if } \mathit{roles}(\hat{S},R^*), R \notin R^* \end{cases}$$

Fig. 8: Projection

`all` is slightly more subtle. If the set being "quantified over" is a role different from the current one, we project under the "quantifier". We do the same if the set is not a role but some data located on R. Otherwise we drop the quantifier. Note that due to the side condition on transmission, there is no use for nesting more than one `all` over the same role. `with` is similar to `all`, except that the set being selected from is not allowed to be a role, as the selection of an arbitrary party cannot be coordinated globally in a simple manner. `if` projects to an `either` depending on whether its condition can be tested by the current role.

To project `while`, a termination condition must be given for the role it is to be projected on, and it is carried through without changes. The body of the loop must not involve roles for which there is no termination condition.

Projection of a statement $\hat{S}$ with respect to a set of roles R^* results in a set of statements $\{\bar{S}_1, ...\}$. A given projection function is *sound* if every behavior of the parallel composition of $\{\bar{S}_1, ...\}$ is a behavior of $\hat{S}$, and *complete* if the converse is true. A sound and complete projection function thus splits a choreography into a set of processes with equivalent behavior. This may be expressed in the following statement, if $(\hat{S}, \sigma) \rightsquigarrow \tau$ denotes that $\hat{S}$ may give rise to the behavior τ from a starting state σ, where τ is a sequence of states.

Theorem 1. *Given* $\hat{S} \upharpoonright R_1 = \bar{S}_1, ...,$ $(\hat{S}, \sigma) \rightsquigarrow \tau \iff (\textit{par}\ \bar{S}_1\ \textit{and}\ ..., \sigma) \rightsquigarrow \tau$.

```
process (p_main \in {main}) {
  fork: await \A p \in {S1, S2} :
    pc[p] = "Done";
}
process (p_S1 \in {S1}) {
  await pc[main] = "fork";
  (* body of S1 *)
}
(* S2 is similar *)
```

Fig. 9: Translation of `par`

```
process (p_main \in {main}) {
  fork:
  await \A p \in {main} \X {p1, p2} :
    pc[p] = "Done"
}
process (p_S \in {main} \X {p1, p2}) {
  await pc[Head(self)] = "fork";
  (* S[Head(self)/v] *)
}
```

Fig. 10: Translation of `all`

5 Translation

Having projected a `choreography` $\hat{S}$ into a set of processes $\bar{S}$, we now cover the translation to Core PlusCal S. The overall translation is denoted by the function $Tr : \bar{S} \rightarrow S$, which is the composition of several functions $Tr_{task} \circ Tr_{par} \circ Tr_{all}$, each handling a specific construct orthogonally. We explain each translation scheme by example and then justify its correctness.

First is Tr_{task}, which handles tasks and cancellation. It instruments models with a global variable `cancelled`, a record whose domain is the set of all task identifiers. For each statement `task` R e $\bar{S}$, `cancelled[e]` is initially `FALSE`. `cancel` e translates to `cancelled`$[e]$:= `TRUE`, and `task` R e $\bar{S}$ by transforming every statement $\bar{S}_1$ in $\bar{S}$ to `if` `cancelled`$[e]$ $\bar{S}_1$ `skip`. Intuitively, this translation is correct because it makes cancellation idempotent and prevents further actions in a task from occurring by making them `skip`s.

The actual statement of semantic preservation is as follows, given that the target language S is a strict subset of the source language $\bar{S}$ and the behaviors produced by both are identical (hence they are equally expressive):

Theorem 2. *Given* $Tr_{task}(\bar{S}) = S$, $(S, \sigma) \rightsquigarrow \tau \iff (Tr_{task}(\bar{S}), \sigma) \rightsquigarrow \tau$.

(`par` $\bar{S}_1$ `and` $\bar{S}_2$) is translated as in Fig. 9. `main` is the process this "fork" is nested in, where control is initially (above). $\bar{S}_1$ and $\bar{S}_2$ expand into separate processes (below) which may only begin execution once `main` reaches the labelled `fork` point. $\bar{S}_1$ and $\bar{S}_2$ execute in parallel, and `main` `await`s their completion at the "join" point before proceeding.

Theorem 3. *Given* $Tr_{par}(\bar{S}) = S$, $(S, \sigma) \rightsquigarrow \tau \iff (Tr_{par}(\bar{S}), \sigma) \rightsquigarrow \tau$.

Next, we cover `all`, which extends the approach (and justification) for `par`. The statement `all` $v \in e$ $\bar{S}$ expands as shown in Fig. 10, assuming $e = \{p_1, p_2\}$.

As before, there is a `main` process with a distinguished `fork` point. It `await`s the completion of all the subprocesses whose identifiers are in e. Now, however, as the value of e may not be statically known, we create a set of processes whose identifiers come from the cartesian product of e and *any roles in enclosing* `all` *statements*; in this case, the only such role is the singleton set `{main}`. Each process body contains $\bar{S}$ with the bound variable v substituted.

This translation scheme is compositional: we can apply it recursively to support arbitrary nesting of `all` statements. This is in contrast to threads in Distributed PlusCal [1], which only allow one level of nesting due to the way program

counters are extended. This gives Choreographic PlusCal much of the flexibility of a process calculus, allowing very direct expression of some protocols.

Theorem 4. *Given* $Tr_{all}(\bar{S}) = S$*,* $(S, \sigma) \rightsquigarrow \tau \iff (Tr_{all}(S), \sigma) \rightsquigarrow \tau$*.*

6 Monitoring

After progressively translating from Choreographic PlusCal to regular PlusCal to TLA$^+$, the next step of the pipeline compiles a TLA$^+$ model into a monitor in an implementation's language. We choose Go here because of its popularity for distributed protocol work, but our overall approach is language-agnostic.

To use a TLA$^+$ model as a monitor for a system, one instruments the system to collect behaviors. Each behavior is checked to see if it satisfies the temporal formula $\mathsf{Init} \land \Box \mathsf{Next}$ of the model. As a testing approach, monitoring is sound but incomplete: failing assertion point to refinement violations and indicate bugs, but nothing can be said about assertions which are never executed.

Specifications of distributed protocols are satisfied by behaviors involving several roles, so traces must be collected from all of them and merged to obtain a behavior to check. This may be inconvenient (requiring events to be ordered using vector clocks) or impractical (if the behaviors of some roles cannot be observed directly). We would thus like to perform monitoring *locally*, checking the behaviors of individual parties in isolation, and deriving global guarantees from the assumption that *all* parties satisfy their specifications.

6.1 Projection of TLA$^+$ Models

This motivates projecting *TLA$^+$ models* – using only variables and actions relevant to a given role to monitor a party of that role. This in turn requires that models have a little more structure, satisfying additional syntactic conditions:

All variables are local to a role or to the Network (a distinguished, singleton role). Variables representing role-local state have record values with the role as domain. All actions are local and have an *initiating* party, intuitively the one which acts, e.g. performing a send, or reacting to a message. Actions take this initiating party as an argument and may modify only its variables. If an action involves communication, it must take the other party as an argument as well; all communication is via message-passing and we assume there is at most one other party. Non-Network actions may only modify variables belonging to the role of the party; Network actions must modify the variables of one other party as well.

Choreographic PlusCal ensures all of these by construction, but handwritten TLA$^+$ models may also satisfy them and thereby be usable as monitors.

With this additional structure, projecting a TLA$^+$ model on a role R leaves only R's actions. Variables for R are straightforwardly projected on the initiating party. Network actions are projected to remove state nonlocal to R.

An example is in Fig. 11. On the left is a model for a set of communicating Nodes. There are two local variables, inbox and outbox, which have record values indexed by node. The set of inflight messages is local to the Network. Nodes can send the string "msg" across the network by putting it in their outbox. The Network delivers messages by moving them from the set of inflight messages into

```
CONSTANT Nodes
VARIABLES inbox, outbox, inflight

Init ==
  /\ inbox = [n \in Nodes |-> {}]
  /\ outbox = [n \in Nodes |-> {}]
  /\ inflight = {}

Send(n) ==
  /\ outbox' = [outbox EXCEPT ![n] =
    outbox[n] \union {"msg"}]

NetworkDeliver(n, m) ==
  /\ inbox' = [inbox EXCEPT ![n] =
    inbox[n] \union {m}]
  /\ inflight' = inflight \ {m}

Next ==
  \/ \E n \in Nodes :
    \/ Send(n)
    \/ \E m \in inflight :
      NetworkDeliver(s, m)
  \/ ...
```

```
CONSTANT Nodes
VARIABLES inbox, outbox

Init ==
  /\ inbox = {}
  /\ outbox = {}

Send(n) ==
  /\ outbox' = outbox \union {"msg"}

NetworkDeliver(n, m) ==
  /\ inbox' = inbox \union {m}
```

Fig. 11: Projection example

some node's inbox. The projection of this model for monitoring the Node role is shown on the right. Local variables are no longer records, but single values for a given node. Non-local variables (inflight) and actions (Next) have vanished. Network actions have been projected by removing conjuncts involving inflight.

6.2 Trace Collection

To facilitate collection of traces, we utilize a deep embedding of TLA$^+$ formulae. This avoids semantic mismatches, e.g. TLA$^+$'s dynamic value types vs Go's static value/interface types. Instrumentation of the system to expose its states as a trace is user-provided, and consists of calls to a function with the following signature, to record the current `State` and the `Action` that led to it.

```
func (m *Monitor) Capture(s State, a Action, args ...TLA)
```

The use of a *refinement mapping* to construct a `State` value from the system state and the placement of `Capture` at appropriate *linearization points*, where snapshots of system state can be observed, are crucial. Because the construction of a `State` value is decoupled from its use, it is possible to make use of arbitrary auxiliary state to build it, or to build it piecemeal if the state of the system cannot be easily snapshot [9]. We assume that this instrumentation produces traces which faithfully represent system behaviors.

6.3 Trace Validation

Checking if a concrete behavior satisfies the model's specification can be done in multiple ways (e.g. model-based trace checking [35, 9]), with different tradeoffs.

Here we experiment with the approach of compiling the model into a monitor (in Go) and executing it on observed behaviors. Each action is interpreted as an assertion on pairs of consecutive states with additional information to make the failure cause clear (the user sees more than just an opaque assertion violation). An artificial example (Fig. 12) illustrates the compilation scheme.

```
                    func CheckA1(prev State, this State, z Int) bool {
                      if !reflect.DeepEqual(prev.x, Int(1)) {
                        return false
A1(z) ==              }
  /\ x = 1            if !reflect.DeepEqual(this.y, Plus(z, Int(2))) {
  /\ y' = z + 2         return false
                      }
                      return true
                    }
```

Fig. 12: TLA$^+$ monitor compilation example

Protocol	**Ch. PlusCal**	**TLA$^+$**
Two-phase commit [24]	23	66
Non-blocking atomic commit [36]	36	96
Raft leader election [33]	46	186

Table 1: Relative specification sizes (LoC)

We benchmark the approach in Sec. 7. Compared to model-based trace checking, monitoring requires more engineering (a TLA$^+$ embedding and translations of required operators) but is more scalable (allowing validation of large numbers of traces from randomized testing) and may be done either online or offline, even enabled in production and used to e.g. block transactions if a bug is detected.

7 Evaluation

We answer the following research questions, divided along two broad axes:

1. What classes of protocols does Choreographic PlusCal express well? How concisely can it render them compared to existing specification languages?
2. How applicable is the monitoring approach to existing codebases? How much overhead does it impose at runtime?

7.1 Expressiveness

Choreographic PlusCal lends itself well to the concise expression of protocols with multiple roles. Our case studies focus on distributed consensus protocols, but protocols from other domains which are typically specified as choreographies or in Alice-and-Bob notation (networking, security) would also benefit.

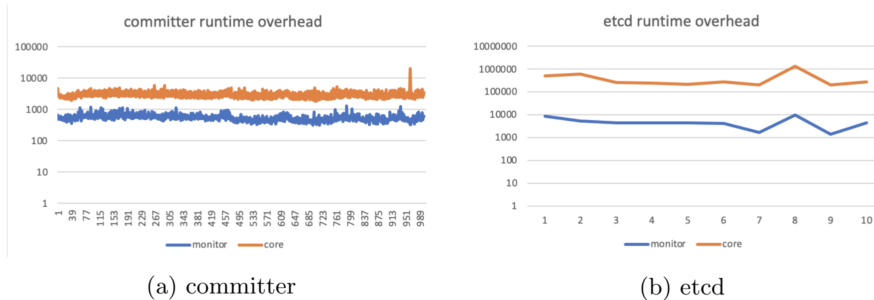
We compare the sizes of such protocol models against handwritten TLA$^+$ in Tab. 1. Writing TLA$^+$ directly offers a great deal of flexibility, but Choreographic PlusCal builds in a lot of structure for the domain and juxtaposes communication, usually halving the sizes of specifications. While LoC is by no means a perfect measure of succinctness and clarity, the point of this comparison is to show that choreographies are an appropriate and promising paradigm for distributed protocols, and make them much easier specify and evolve.

7.2 Monitoring

To evaluate the scalability of the monitoring approach, we instrumented two systems to check their conformance to their protocols. *committer* was the largest implementation under the `two-phase-commit` GitHub tag in Go, while *etcd* is a

Project	Protocol	LoC	Overhead
vadiminshakov/committer	2PC	3032	19% (5 ms)
etcd-io/raft	Raft leader election	21,064	2% (4 ms)

Table 2: Monitor overhead



(a) committer

(b) etcd

Fig. 13: Runtime overhead measurements

key infrastructure piece for platforms such as Kubernetes. Neither project was designed for monitoring, and the models used to monitor them were taken from the public TLA$^+$ examples repository and were not written with them in mind.

For *committer*, we issued 1000 requests to write different keys (and trigger its commit protocol) in a blocking manner. For *etcd*, we waited for a cluster of newly created nodes to elect a leader and stabilize, repeating this 10 times. The overhead figures in Tab. 2 (distribution in Fig. 13, where the x-axis is the request number and the y-axis is time taken) represent the percentage (and absolute amount) of time spent in monitor code over the amount of time spent in their core consensus flows, per request. Experiments were carried out in clusters of size 3 (one coordinator and two replicas, and 3 servers), on a 2020 MacBook Pro with a 2.3 GHz CPU and 16 GB RAM, with Go 1.19.4. The runtime overhead was consistently low (up to two orders of magnitude for etcd), making monitoring readily usable for randomized testing and CI, possibly even in production as a sort of sanitizer. Monitoring also performs significantly better than model-based trace checking with a separate TLC process [9, 35]; I/O for such a task typically adds hundreds of ms, compared to 4-5 ms that in-process monitoring costs.

8 Related Work

8.1 Protocol Specifications

Protocols are classically specified as state machines [21] or automata [26], and early specification languages such as TLA$^+$ [22] and IOA [13] provided a concrete syntax for defining such models. While they are simple to manipulate symbolically, such specifications are unstructured (Sec. 1, Sec. 2) and are hard to extend and comprehend as they grow. Despite that, they are popular as a classical approach for embedding in general-purpose languages [18, 4, 11, 12] and proof assistants [40, 37, 17, 2], supporting extraction validation by model checking, testing, symbolic execution, or deductive verification.

Since protocols are essentially distributed algorithms which compute and branch, they are also naturally specified as imperative programs, in the spirit of "exhaustively testable pseudocode" [28]. DistAlgo [25] extends Python for this, adding queries over message history. Ivy [34] uses carefully constrained imperative programs to enable decidable reasoning. PlusCal [23], Distributed PlusCal [1] (which adds threads, channels, and communication primitives such as multicasts) and Modular PlusCal [43, 8] (which separates system descriptions from their environment to support program extraction) are also in this category.

Choreographies make up a third family of specification styles, describing interactions globally and utilizing projection. Examples are Choral [14], a language for writing executable descriptions of protocols, and session-typed systems [15, 10], many of which utilize monitoring [29, 31, 30, 5, 3].

Our work aims to combine the benefits of these three approaches. By integrating choreographies into PlusCal, a system for which transparency of the compilation to TLA$^+$ model is an explicit goal, we are able to support succinct and high-level specifications which also have a clear semantics, and for practical goals of model checking and conformance testing.

8.2 Testing Distributed Systems

We distinguish distributed systems testing approaches which require clean-room implementations [18, 25, 11, 12, 43] or extensive generated scaffolding [14, 42] from those which aim to validate existing systems with minimal changes, like ours. Dealing with existing systems not originally designed with verification in mind is challenging, and falling back on testing is the usual compromise. Jepsen [19] is a well-known system for black-box testing of distributed databases. It exemplifies the offline monitoring approach: implementation traces are recorded, then analyzed later for indications of bugs using correctness conditions such as linearizability, serializability, and transaction isolation [20]. Model-based trace checking [16] is another similarly offline approach; specializations to TLA$^+$ [35, 9] use TLC to determine conformance with a model. Online approaches include SPL [27], which uses imperative event-based specifications for monitoring, model-based test case generation [38, 9], and stateless model checking [41].

9 Conclusion

We proposed Choreographic PlusCal, a suite of extensions to PlusCal to enable more abstract and succinct specifications of communication-centric algorithms. We hope that it may become for distributed protocols what PlusCal was to shared-memory concurrent algorithms. Separately, we extended the TLA$^+$ tools to enable monitoring implementations for protocol conformance, resulting in a practical and scalable approach for improving the robustness of existing systems.

Acknowledgments. We would like to thank Ilya Sergey for the insights that led to this work, George Pîrlea for contributing to an early implementation of it, and Markus Kuppe, Leslie Lamport, and the anonymous reviewers for their thoughtful suggestions and comments.

Bibliography

[1] H. Alkayed, H. Cirstea, and S. Merz. An extension of pluscal for modeling distributed algorithms. In *TLA+ Community Event 2020*, 2020.

[2] A. Athalye. *CoqIOA: a formalization of IO automata in the Coq proof assistant*. PhD thesis, Massachusetts Institute of Technology, 2017.

[3] C. Bartolo Burlò, A. Francalanza, and A. Scalas. 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] M. Biely, P. Delgado, Z. Milosevic, and A. Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *International Conference on Dependable Systems and Networks (DSN)*, pages 1–8. IEEE, 2013.

[5] L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *Formal Techniques for Distributed Systems*, pages 50–65. Springer, 2013.

[6] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, et al. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 836–850, 2021.

[7] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.

[8] R. M. Costa. *Compiling distributed system specifications into implementations*. PhD thesis, University of British Columbia, 2019.

[9] A. Davis, M. Hirschhorn, and J. Schvimer. Extreme modelling in practice. *arXiv preprint arXiv:2006.00915*, 2020.

[10] P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 435–446, 2011.

[11] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices*, 48(6):321–332, 2013.

[12] A. Desai, A. Phanishayee, S. Qadeer, and S. A. Seshia. Compositional programming and testing of dynamic distributed systems. (OOPSLA):1–30, 2018.

[13] S. J. Garland, N. A. Lynch, and M. Vaziri. IOA: A language for specifying, programming, and validating distributed systems. *Unpublished manuscript*, 1997.

[14] S. Giallorenzo, F. Montesi, and M. Peressotti. Choreographies as objects. *arXiv*.

[15] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.

[16] Y. Howard, S. Gruner, A. Gravell, C. Ferreira, and J. C. Augusto. Model-based trace-checking. *arXiv preprint arXiv:1111.2825*, 2011.

[17] C. Hsieh and S. Mitra. Dione: A protocol verification system built with dafny for i/o automata. In *International Conference on Integrated Formal Methods*, pages 227–245. Springer, 2019.

[18] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. *ACM Sigplan Notices*, pages 179–188, 2007.

[19] K. Kingsbury. A framework for distributed systems verification, with fault injection, 2022.

[20] K. Kingsbury and P. Alvaro. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554*, 2020.

[21] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[22] L. Lamport. *Specifying systems*, volume 388. Addison-Wesley Boston, 2002.

[23] L. Lamport. The pluscal algorithm language. In *International Colloquium on Theoretical Aspects of Computing*, pages 36–60. Springer, 2009.

[24] B. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. 1979.

[25] Y. A. Liu, S. D. Stoller, B. Lin, and M. Gorbovitski. From clarity to efficiency for distributed algorithms. Number OOPSLA, pages 395–410, 2012.

[26] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 137–151, 1987.

[27] A. Madhavapeddy. Combining static model checking with dynamic enforcement using the statecall policy language. In *International Conference on Formal Engineering Methods*, pages 446–465. Springer, 2009.

[28] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, pages 66–73, 2015.

[29] R. Neykova, L. Bocchi, and N. Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects of Computing*, 29(5):877–910, 2017.

[30] R. Neykova and N. Yoshida. Multiparty session actors. In *International Conference on Coordination Languages and Models*, pages 131–146. Springer, 2014.

[31] R. Neykova and N. Yoshida. Let it recover: multiparty protocol-induced recovery. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 98–108, 2017.

[32] D. Ongaro. Tla+ specification for the raft consensus algorithm, 2022.

[33] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX*, pages 305–319, 2014.

[34] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630, 2016.

[35] R. Pressler. Verifying software traces against a formal specification with tla+ and tlc. 2018.

[36] M. Raynal. A case study of agreement problems in distributed systems: non-blocking atomic commitment. In *HASE*, pages 209–214, 1997.

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

[38] T. Tervoort and I. Prasetya. Modeling and testing implementations of protocols with complex messages. *arXiv preprint arXiv:1804.03927*, 2018.

[39] tlaplus. A collection of tla+ specifications of varying complexities, 2022.

[40] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

[41] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. 2009.

[42] N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In *International Symposium on Trustworthy Global Computing*, pages 22–41, 2013.

[43] B. Zhang. Pgo: Corresponding a high-level formal specification with its implementation. *SOSP SRC*, page 3, 2016.

A Auxiliary Definitions

Definition 3 (Local variables and expressions). *Declared local variables v of parties of a role R are said to be* local to R*, written $loc(v, R)$. An expression e is* local to R *iff $loc(v, R)$ holds for all variables v in e and all other values in e are constants; this is denoted $loc(e, R)$.*

Definition 4 (Party and role expressions). *An expression e is a* role expression *if it evaluates to a subset of some role R, and a* party expression *if it evaluates to a* member *of R. These are written $role(e, R)$ and $party(e, R)$.*

Definition 5 (Task roles). *$task(e, R)$ holds if there is a task $\texttt{task}\ R\ e\ S$.*

Definition 6 (Initiative). *Relates a global statement S and the next role to act R as it reduces. R is said to have the initiative in S if $initiative(S, R)$ holds.*

$$
\begin{aligned}
initiative(v := e, R) &\ \text{iff}\ loc(e, R) \\
initiative(\texttt{await}\ e, R) &\ \text{iff}\ loc(e, R) \\
initiative(\texttt{while}\ e\ S, R) &\ \text{iff}\ loc(e, R) \\
initiative(\texttt{with}\ v \in e\ S, R) &\ \text{iff}\ loc(e, R) \\
initiative(\texttt{all}\ v \in e\ S, R) &\ \text{iff}\ loc(e, R) \\
initiative(\texttt{par}\ S_1\ \texttt{and}\ S_2, R) &\ \text{iff}\ initiative(S_1, R) \\
initiative(S_1;\ S_2, R) &\ \text{iff}\ initiative(S_1, R) \\
initiative(\texttt{Transmit}(s, r, v = m), R) &\ \text{iff}\ party(s, R) \\
initiative(\texttt{Send}(s, r, m), R) &\ \text{iff}\ party(s, R) \\
initiative(v := \texttt{Receive}(s, r), R) &\ \text{iff}\ party(r, R) \\
initiative(\texttt{task}\ Q\ e\ S, R) &\ \text{iff}\ initiative(Q, R) \\
initiative(\texttt{cancel}\ e, R) &\ \text{iff}\ task(e, R)
\end{aligned}
$$

Definition 7 (Syntactic membership). *Given $party(self, R)$ and $role(e, R)$, $self \in e$ holds if $self \in R$.*

Definition 8 (Structural congruence).

$$\hat{S} \equiv \hat{S} \quad \texttt{par}\ \hat{S}_1\ \texttt{skip}\ \hat{S}_2\ \texttt{and} \equiv \hat{S}_2 \quad \texttt{par}\ \hat{S}_1 \quad \texttt{and} \quad \texttt{par} \quad \texttt{and}$$

$$(\quad \hat{S}_1 \quad \hat{S}_2) \quad \hat{S}_3 \equiv \texttt{par}\ \texttt{par}\ \hat{S}_1\ \texttt{and}\ (\quad \hat{S}_2\ \texttt{and} \quad \hat{S}_3)\ \texttt{par} \quad \texttt{and}\ \texttt{par} \quad \texttt{and}$$

$$v \in e \quad \equiv \quad \texttt{all}\ v \in \{\}\ \texttt{skip}\ \hat{S} \equiv \quad \texttt{skip} \quad \texttt{all} \quad \texttt{skip}$$

$$R\ e \quad \equiv \quad \texttt{task} \quad \texttt{skip} \quad \texttt{skip}$$

$$\hat{S}_1 \quad \hat{S}_2 \equiv \quad \texttt{either}\ \hat{S}_2\ \hat{S}_1 \quad \texttt{or} \quad \texttt{either} \quad \texttt{or}$$

$$(\quad \hat{S}_1 \quad \hat{S}_2)\ \texttt{either}\ \hat{S}_3 \equiv \texttt{either}\ \hat{S}_1 \quad \texttt{or}\ (\quad \texttt{or}\ \hat{S}_2\ \texttt{either}\ \hat{S}_3) \quad \texttt{or}\ \texttt{either} \quad \texttt{or}$$

$$;\ \hat{S} \equiv \hat{S} \quad \hat{S}; \quad \equiv \texttt{skip}\ \hat{S} \quad (\hat{S}_1;\ \hat{S}_2);\ \hat{S}_3 \equiv \texttt{skip}\ \hat{S}_1;\ (\hat{S}_2;\ \hat{S}_3)$$

$$\hat{S}_1;\ \hat{S}_2 \equiv \hat{S}_2;\ \hat{S}_1\ \ \textit{if}\ roles(\hat{S}_1, R_1^*), roles(\hat{S}_2, R_2^*), R_1^* \cap R_2^* = \{\}$$

$$
\begin{array}{rll}
(v := e, \sigma) \rightarrow (\texttt{skip}, \sigma[e_1/v]) & & \text{if } [[e]]\sigma = e_1 \\
(\texttt{await } e, \sigma) \rightarrow (\texttt{skip}, \sigma) & & \text{if } [[e]]\sigma \\
(\texttt{await } e, \sigma) \rightarrow (\texttt{await } e, \sigma) & & \text{otherwise} \\
(\texttt{while } e\ \hat{S}, \sigma) \rightarrow (\hat{S};\ \texttt{while } e\ \hat{S}, \sigma) & & \text{if } [[e]]\sigma \\
(\texttt{while } e\ \hat{S}, \sigma) \rightarrow (\texttt{skip}, \sigma) & & \text{otherwise} \\
(\texttt{if } e\ \hat{S}_1\ \hat{S}_2, \sigma) \rightarrow (\hat{S}_1, \sigma) & & \text{if } [[e]]\sigma \\
(\texttt{if } e\ \hat{S}_1\ \hat{S}_2, \sigma) \rightarrow (\hat{S}_2, \sigma) & & \text{otherwise} \\
(\texttt{either } \hat{S}_1 \texttt{ or } \hat{S}_2, \sigma) \rightarrow (\hat{S}_1, \sigma) & & \\
(\texttt{either } \hat{S}_1 \texttt{ or } \hat{S}_2, \sigma) \rightarrow (\hat{S}_2, \sigma) & & \\
(\texttt{with } v \in e\ \hat{S}, \sigma) \rightarrow (\hat{S}, \sigma[e_1/v]) & & \text{if } [[e]]\sigma = \{e_1, ...\} \\
(\texttt{all } v \in e\ \hat{S}, \sigma) \rightarrow (\texttt{par } \hat{S}[e_1/v] \texttt{ and } ..., \sigma) & & \text{if } [[e]]\sigma = \{e_1, ...\} \\
(\texttt{par } \hat{S}_1 \texttt{ and } \hat{S}_2, \sigma) \rightarrow (\texttt{par } \hat{S}_1' \texttt{ and } \hat{S}_2, \sigma') & & \text{if } (\hat{S}_1, \sigma) \rightarrow (\hat{S}_1', \sigma') \\
(\hat{S}_1;\ \hat{S}_2, \sigma) \rightarrow (\hat{S}_1';\ \hat{S}_2, \sigma') & & \text{if } (\hat{S}_1, \sigma) \rightarrow (\hat{S}_1', \sigma') \\
(\texttt{task } R\ e\ \hat{S}, \sigma) \rightarrow (\texttt{task } R\ e\ \hat{S}', \sigma') & & \text{if } (\hat{S}, \sigma) \rightarrow (\hat{S}', \sigma'), \\
 & & \quad \textit{initiative}(\hat{S}, R), \text{not } [[v_e]]\sigma \\
(\texttt{task } R\ e\ \hat{S}, \sigma) \rightarrow (\texttt{task } R\ e\ \hat{S}', \sigma) & & \text{if } (\hat{S}, \sigma) \rightarrow (\hat{S}', \sigma'), \\
 & & \quad \textit{initiative}(\hat{S}, R), [[v_e]]\sigma \\
(\texttt{task } R\ e\ \hat{S}, \sigma) \rightarrow (\texttt{task } R\ e\ \hat{S}', \sigma') & & \text{if } (\hat{S}, \sigma) \rightarrow (\hat{S}', \sigma'), \\
 & & \quad \textit{initiative}(\hat{S}, Q), Q \neq R \\
(\texttt{cancel } v_e, \sigma) \rightarrow (\texttt{skip}, \sigma[\texttt{TRUE}/v_e]) & & \\
(\texttt{Transmit}(s, r, v = m), \sigma) \rightarrow (\texttt{Send}(s, r, m);\ v := \texttt{Receive}(s, r), \sigma') & &
\end{array}
$$

$$
\frac{}{(\quad, \sigma) \leadsto []} \qquad \frac{(\hat{S}, \sigma) \rightarrow (\hat{S}', \sigma') \qquad (\hat{S}', \sigma') \leadsto \tau}{(\hat{S}\texttt{skip}, \sigma) \leadsto \sigma :: \tau} \qquad \tau ::= [] \mid \sigma :: \tau
$$

Fig. 14: Operational Semantics of Choreographic PlusCal

B Operational Semantics of Choreographic PlusCal

The operational semantics of Choreographic PlusCal is shown in Fig. 14. It is inductively defined as a small-step reduction relation $\rightarrow$ between *configurations* (statement-state pairs), modulo structural congruence (Def. 8).

The rules clarify the correspondences between the different statements: `await` blocks until the condition is true whereas `while` loops until it is false, `with` executes the body with exactly one element of e while `all` does so for every element, `if` and `either` branch conditionally or nondeterministically, and sequential and parallel composition simply execute one of their child statements.

A task propagates inward as its body executes. It depends on an auxiliary predicate, *initiative* (Def. 6), which relates statements and the role that their *next* reduction involves; intuitively *initiative* switches back and forth between roles. The effect of the task body is retained if it has not been cancelled or if its role does not have initiative, otherwise its effect is ignored. Cancellation sets a distinguished global variable indexed by the task label, which subsequent task executions may use to tell if they have been cancelled.

It is worth noting that sequential composition at the global level may express unenforceable orderings (via the last structural congruence rule); consider for example the program $(Transmit(a, b, v_1 = m_1); Transmit(c, d, v_2 = m_2))$, where all parties are from different roles.

We assume `Send` and `Receive` are user-provided functions with the following semantics: `Send` buffers a message to the given recipient, and `Receive` returns the buffered message of *any* prior `Send` from the given sender, blocking if it is not available, and assigning to the given variable v.

The definitions make use of the auxiliary relation *roles* (Def. 4) which intuitively relates a statement to the set of roles involved in its execution.

Finally, $(\hat{S}, \sigma_0) \rightsquigarrow \tau$ holds if a given starting configuration $(\hat{S}, \sigma_0)$ may give rise to the behavior τ. The set of behaviors of a Choreographic PlusCal fragment $\hat{S}$ is thus defined to be $b(\hat{S}) = \{\ \tau \mid (\hat{S}, \sigma_0) \rightsquigarrow \tau\ \}$. This may be an infinite set due to the presence of loops. Each behavior in the set, however, is of arbitrary but finite length; we assume loops are productive and protocols have a termination condition. Reasoning about nonterminating protocols is left to future work.

C Proofs (Translation)

Theorem 5. *Given* $Tr_{task}(\bar{S}) = S$*,* $(S, \sigma) \rightsquigarrow \tau \iff (Tr_{task}(\bar{S}), \sigma) \rightsquigarrow \tau$*.*

Proof. We proceed by induction on the structure of $\bar{S}$. As Tr_{task} only transforms two language constructs, the proof for all the other cases is immediate. $Tr_{task}($`cancel` $e)$ is practically identical to its definition in Fig. 14, and both versions produce an identical, singleton trace with a distinguished variable for e set to `TRUE`. Next, we consider how $Tr_{task}($`task` $R\ e\ \bar{S}_1)$ executes in each of the three cases in Fig. 14. By the IH, we have that the translation of $\bar{S}_1$ preserves its semantics, so it remains to show for each case that the source and target behave equivalently under a single reduction step. If the task has not been cancelled and R has the initiative, `task` $R\ e\ \bar{S}_1$ reduces to `task` $R\ e\ \bar{S}_1'$; this is equivalent to the topmost `if` in $Tr_{task}($`task` $R\ e\ \bar{S}_1)$ *not* being a no-op and reducing to S_1'; as child statements all have the same checks, it then reduces as `task` $R\ e\ \bar{S}_1'$ would. If it has been cancelled, the topmost `if` results in a no-op; this is the same as a single reduction with no effect. If R does not have the initiative, cancellation has no effect, and thus in all cases the source and target behaviors are equivalent.

Theorem 6. *Given* $Tr_{par}(\bar{S}) = S$*,* $(S, \sigma) \rightsquigarrow \tau \iff (Tr_{par}(\bar{S}), \sigma) \rightsquigarrow \tau$*.*

Proof. By induction on the structure of $\bar{S}$, focusing on the case of parallel composition. Consider (`par` $\bar{S}_1$ `and` $\bar{S}_2, \sigma) \rightsquigarrow \tau$. By the IH, the property holds for $\bar{S}_1$ and $\bar{S}_2$, and τ is some interleaving of their behaviors. $Tr_{par}(\bar{S}_1)$ and $Tr_{par}(\bar{S}_2)$ may give rise to τ as well, as they result in separate `process`es with identical interleaving semantics.

Theorem 7. *Given* $Tr_{all}(\bar{S}) = S$*,* $(S, \sigma) \rightsquigarrow \tau \iff (Tr_{all}(S), \sigma) \rightsquigarrow \tau$*.*

Proof (Sketch.). From Fig. 14, `all` is semantically equivalent to the parallel composition of its body $|e|$ times. Each nested `all` statement multiplicatively increases the number of parallel compositions, thus the number of threads in the innermost `par` is the product of the sizes of all the sets. This is exactly what the translation to `process`es identified by the cartesian product of all nested sets expresses.

D Reflexive Projection

The restriction in Sec. 4 that parties of a single role do not send messages to each other, does not seem severe at first. However, many protocols have only a single conceptual role, e.g. Chang and Roberts' leader election in a ring [7] and non-blocking atomic commit [36] (Sec. E.1). Other protocols have entirely dynamic roles (and thus a single static *role*, in our sense of the word), e.g. Raft [33] (Sec. E.2), where node *states* (e.g. *follower*, *candidate*, and *leader*) determine their pattern of behavior.

We call such protocols *dynamic multirole protocols*, following Deniélou et al. [10]. They require a more granular projection function $\upharpoonright^+$ which is sensitive to self-sends. This new function changes only two cases, shown in Fig. 15.

In the case for `Transmit`, there is the additional possibility that s and r belong to the same role; in that case, we produce *both* a send and a receive. The `all` case is more involved. Projection is now defined with respect to a role R and an arbitrary member of that role. We use the distinguished value *self* as a syntactic proxy for that party; $\mathit{party}(\mathit{self}, R)$ thus holds. The first case is the same as before: if the set e may be interpreted as another role, the quantifier pertains neither to the current role nor party and thus should stay; we project under it. Otherwise, we must check the syntactic condition $\mathit{self} \in e$ (Def. 7); $\mathit{self} \in e$ holds, given $\mathit{self} \in R$ and $\mathit{role}(e, R)$, and $\mathit{self} \in (e \setminus \{self\})$ does not. In the second case, if $\mathit{self} \in e$, we must account for the possibility of a self-send. This produces two parallel behaviors, one for what *self* does, and another for what *the other parties in its role do*. The result in the final case is the same as in the first; since *self* is not in e, we need only account for what other parties do. An example of what this projection function produces is shown in Sec. D.1.

Theorem 8. *Given* $\hat{S} \upharpoonright^+ R_1 = \bar{S}_1, ..., (\hat{S}, \sigma) \rightsquigarrow \tau \iff (\textit{par}\ \bar{S}_1\ \textit{and}\ ..., \sigma) \rightsquigarrow \tau$.

$$\texttt{Transmit}(s, r, v = m) \upharpoonright^+ R \triangleq \begin{cases} \texttt{Send}(s, r, m);\ v := \texttt{Receive}(s, r) & \text{if } \mathit{party}(s, R), \mathit{party}(r, R) \\ \texttt{Send}(s, r, m) & \text{if } \mathit{party}(r, R) \\ v := \texttt{Receive}(s, r) & \text{if } \mathit{party}(s, R) \\ \texttt{skip} & \text{otherwise} \end{cases}$$

$$(\texttt{all}\ v \in e\ \hat{S}) \upharpoonright^+ R \triangleq \begin{cases} \texttt{all}\ v \in e\ (\hat{S} \upharpoonright^+ R) & \text{if } \mathit{role}(e, Q), Q \neq R \\ \texttt{par}\ (\hat{S}[\mathit{self}/v] \upharpoonright^+ R)\ \texttt{and}\ (\texttt{all}\ v \in (e \setminus \{\mathit{self}\})\ (\hat{S} \upharpoonright^+ R)) & \text{if } \mathit{self} \in e \\ \texttt{all}\ v \in e\ (\hat{S} \upharpoonright^+ R) & \text{otherwise} \end{cases}$$

Fig. 15: Extended Projection

D.1 Reflexive Projection Example

The following (artificial) program:

```
choreography
  (x \in C)
    variables a = "start", b = "start";
{
  all (c \in C) {
    all (d \in C) {
      Transmit(c, d, v = m);
      a[d] = "y";
      b[c] = "z";
    }
  }
}
```

projects to:

```
choreography
  (x \in C)
    variables a = "start", b = "start";
{
  par {
    Send(self, m);
    Receive(self, m);
    a = 1;
    b = 2;
  } and {
    all (d \in (C \ {self})) {
      Send(d, m);
      b = 2;
    }
  } and {
    all (c \in (C \ {self})) {
      v := Receive(c);
      a = 1;
    }
  }
}
```

E Case Studies

E.1 Nonblocking Atomic Commit

Nonblocking Atomic Commit (NBAC) [36] is a classic refinement of two-phase commit and a precursor to Paxos. To avoid centralizing the decision of whether to commit (and blocking if that central coordinator fails), NBAC enables all parties to take responsibility for the decision to commit. There is thus just one set of "processes" P (which we call *participants*, to avoid overloading the term). Each sends a vote on whether to commit to every other process, *including itself*, and tallies votes upon receiving them. This is naturally expressed in lines 10-19 of Fig. 16. The next part (lines 21-29) expresses when a commit occurs: when all processes have voted yes, or when some process has voted no or failed.

NBAC relies on a failure detector (lines 31-36) that informs each process `p` of the failure of *another process* `q`. This provides the condition for protocol to end.

```
1  choreography
2    (F \in failure_detectors)
3    (P \in participants)
4      variables
5        voted_yes = {},
6        voted_no = FALSE,
7        outcome = "none"; {
8    par {
9      task participants "votes" {
10       all (p \in participants) {
11         all (q \in participants) {
12           either {
13             Transmit(p, q, "yes");
14             voted_yes[q] = voted_yes[q] \cup {p};
15           } or {
16             Transmit(p, q, "no");
17             voted_no[q] = TRUE;
18             cancel "votes";
19           }  }  }  }
20   } and {
21     all (p \in participants) {
22       either {
23         await voted_no \/ some_failed;
24         outcome := "abort";
25       } or {
26         await voted_yes = participants;
27         outcome := "commit";
28       }
29     }
30   } and {
31     all (f \in failure_detectors) {
32       all (p \in participants) {
33         all (q \in participants \ {p}) {
34           Transmit(f, p, <<"failed", q>>)
35           some_failed[p] := TRUE;
36         }  }  }  }  }
```

Fig. 16: NBAC

E.2 Raft

Raft [33] is a modern and comprehensive take on consensus. It is a dynamic multirole protocol: parties begin as *followers*, and upon a random timeout, they transition to *candidates* and vie to become *leader* via an election. Upon receiving a quorum of votes, a party declares itself leader and begins to service requests, replicating them to other parties. As Raft is fully-featured, we show only the leader election fragment in Fig. 17, with details left out for simplicity.

We express Raft as the parallel composition of a number of subprotocols. The first controls timeouts (line 6): a party may time out at any point, continuously, as long as it is a follower. Once it becomes a candidate, it may start an election (line 12), which also occurs continuously as long as it is not in the middle of an existing election.

During the election it sends `RequestVote` messages to other parties, including itself (line 21). Recipients decide whether to `grant` requests, track who they voted for, and reply (lines 26-33). If the reply is positive and not stale, the leader tallies it (line 37). It also waits for a quorum (line 40), or give up waiting and begin a new election (line 43).

The original TLA$^+$ specification of Raft [32] represents each of these steps as individual actions, using guards to control when they may occur (the rough equivalent in our language would be writing each step as a recursive `protocol` and composing them all in parallel). It is lower-level, and also more permissive: for example, a leader is allowed to time out after sending any number of `RequestVote` messages, despite the fact that an implementation would never do this. Admitting such behaviors is conceptually harmless, but are redundant and not useful when running a model checker. This is something we are able to capture easily with a more structured specification, and something we are more sensitive to given the goal of instrumenting implementations.

```
1  choreography
2    (S \in servers)
3      variables role = "follower", votes = {},
4        voted_for = <<>>, term = 0, grant = FALSE, ...; {
5    par { \* Handle timeouts
6      all (s \in servers) {
7        while (TRUE) {
8          await role = "follower";
9          role := "candidate";
10       }  }
11   } and { \* Start an election
12     all (s \in servers) {
13       while (TRUE) {
14         await role = "candidate";
15         votes := {};
16         term := term + 1;
17         par {
18           task servers s {
19             \* Vote for ourselves too
20             all (t \in servers) {
21               Transmit(s, t, msg = [
22                 Type |-> "RequestVote",
23                 Term |-> term[s]]);
24               \* Drop stale messages
25               if (msg.Term <= term[t]) {
26                 grant[t] := ...;
27                 if (grant[t]) {
28                   voted_for[t] = <<t>>;
29                 }
30                 Transmit(t, s, [
31                   Type |-> "RequestVoteResp",
32                   Term |-> term[t],
33                   Grant |-> grant[t]]);
34                 \* Drop stale messages
35                 if (msg.Term = term[s]) {
36                   if (msg.Granted) {
37                     votes[s] := votes[s] \union {t};
38                   }  }  }  }  }
39         } and {
40           await Cardinality(votes) > Quorum(servers);
41           role := "leader";
42         } and {
43           cancel s;
44         }
45       } \* end while
46     } } and { ... } }
```

Fig. 17: Raft leader election

F Two-phase Commit in PlusCal and TLA$^+$

```
-------------------- MODULE TwoPhaseCommit ---------------------
EXTENDS Naturals, TLC, Sequences

CONSTANTS p1, p2, coord

(* --algorithm TwoPhaseCommit {
  variables
    participants = {p1, p2};
    messages = {};

  macro Send(from, to, type) {
    messages := messages \union {[To |-> to, From |-> from, Type |-> type]};
  }

  macro Receive(from, to, type) {
    await [To |-> to, From |-> from, Type |-> type] \in messages;
  }

  process (P \in participants)
  {
    Receive(coord, self, "prepare");

    either {
    psend:
      Send(self, coord, "prepared");
    } or {
      Send(self, coord, "abort");
    };

    either {
      Receive(coord, self, "commit");
      Send(self, coord, "committed");
    } or {
      Receive(coord, self, "abort");
      Send(self, coord, "aborted");
    };
  };

  process (Coordinator = coord)
    variables temp = participants, aborted = FALSE;
  {
      while (temp /= {}) {
        with (r \in temp) {
          Send(coord, r, "prepare");
          temp := temp \ {r};
        };
      };

      temp := participants;
      while (temp /= {} \/ aborted) {
        with (r \in temp) {
          either {
            Receive(r, coord, "prepared");
          } or {
            Receive(r, coord, "aborted");
            aborted := TRUE;
          };
          temp := temp \ {r};
        };
      };

      if (aborted) {
        temp := participants;
        while (temp /= {}) {
          with (r \in temp) {
```

```
            Send(coord, r, "abort");
            temp := temp \ {r};
          };
        };

        temp := participants;
        while (temp /= {}) {
          with (r \in temp) {
            Receive(r, coord, "aborted");
            temp := temp \ {r};
          };
        };
      } else {
        temp := participants;
        while (temp /= {}) {
          with (r \in temp) {
            Send(coord, r, "commit");
            temp := temp \ {r};
          };
        };

        temp := participants;
        while (temp /= {}) {
          with (r \in temp) {
            Receive(r, coord, "committed");
            temp := temp \ {r};
          };
        };
      };
  }

}
*)
\* BEGIN TRANSLATION (chksum(pcal) = "a7969572" /\ chksum(tla) = "2ef5998a")
VARIABLES participants, messages, pc, temp, aborted

vars == << participants, messages, pc, temp, aborted >>

ProcSet == (participants) \cup {coord}

Init == (* Global variables *)
        /\ participants = {p1, p2}
        /\ messages = {}
        (* Process Coordinator *)
        /\ temp = participants
        /\ aborted = FALSE
        /\ pc = [self \in ProcSet |-> CASE self \in participants -> "Lbl_1"
                                        [] self = coord -> "Lbl_3"]

Lbl_1(self) == /\ pc[self] = "Lbl_1"
               /\ [To |-> self, From |-> coord, Type |-> "prepare"] \in
                   messages
               /\ \/ /\ pc' = [pc EXCEPT ![self] = "psend"]
                     /\ UNCHANGED messages
                  \/ /\ messages' = (messages \union {[To |-> coord, From |->
                        self, Type |-> "abort"]})
                     /\ pc' = [pc EXCEPT ![self] = "Lbl_2"]
               /\ UNCHANGED << participants, temp, aborted >>

psend(self) == /\ pc[self] = "psend"
               /\ messages' = (messages \union {[To |-> coord, From |-> self,
                    Type |-> "prepared"]})
               /\ pc' = [pc EXCEPT ![self] = "Lbl_2"]
               /\ UNCHANGED << participants, temp, aborted >>

Lbl_2(self) == /\ pc[self] = "Lbl_2"
               /\ \/ /\ [To |-> self, From |-> coord, Type |-> "commit"] \in
                   messages
```

```
                  /\ messages' = (messages \union {[To |-> coord, From |->
                       self, Type |-> "committed"]})
               \/ /\ [To |-> self, From |-> coord, Type |-> "abort"] \in
                     messages
                  /\ messages' = (messages \union {[To |-> coord, From |->
                       self, Type |-> "aborted"]})
            /\ pc' = [pc EXCEPT ![self] = "Done"]
            /\ UNCHANGED << participants, temp, aborted >>

P(self) == Lbl_1(self) \/ psend(self) \/ Lbl_2(self)

Lbl_3 == /\ pc[coord] = "Lbl_3"
         /\ IF temp /= {}
               THEN /\ \E r \in temp:
                          /\ messages' = (messages \union {[To |-> r, From |->
                               coord, Type |-> "prepare"]})
                          /\ temp' = temp \ {r}
                    /\ pc' = [pc EXCEPT ![coord] = "Lbl_3"]
               ELSE /\ temp' = participants
                    /\ pc' = [pc EXCEPT ![coord] = "Lbl_4"]
                    /\ UNCHANGED messages
         /\ UNCHANGED << participants, aborted >>

Lbl_4 == /\ pc[coord] = "Lbl_4"
         /\ IF temp /= {} \/ aborted
               THEN /\ \E r \in temp:
                          /\ \/ /\ [To |-> coord, From |-> r, Type |-> "
                                  prepared"] \in messages
                                /\ UNCHANGED aborted
                             \/ /\ [To |-> coord, From |-> r, Type |-> "
                                     aborted"] \in messages
                                /\ aborted' = TRUE
                          /\ temp' = temp \ {r}
                    /\ pc' = [pc EXCEPT ![coord] = "Lbl_4"]
               ELSE /\ IF aborted
                           THEN /\ temp' = participants
                                /\ pc' = [pc EXCEPT ![coord] = "Lbl_5"]
                           ELSE /\ temp' = participants
                                /\ pc' = [pc EXCEPT ![coord] = "Lbl_7"]
                    /\ UNCHANGED aborted
         /\ UNCHANGED << participants, messages >>

Lbl_5 == /\ pc[coord] = "Lbl_5"
         /\ IF temp /= {}
               THEN /\ \E r \in temp:
                          /\ messages' = (messages \union {[To |-> r, From |->
                               coord, Type |-> "abort"]})
                          /\ temp' = temp \ {r}
                    /\ pc' = [pc EXCEPT ![coord] = "Lbl_5"]
               ELSE /\ temp' = participants
                    /\ pc' = [pc EXCEPT ![coord] = "Lbl_6"]
                    /\ UNCHANGED messages
         /\ UNCHANGED << participants, aborted >>

Lbl_6 == /\ pc[coord] = "Lbl_6"
         /\ IF temp /= {}
               THEN /\ \E r \in temp:
                          /\ [To |-> coord, From |-> r, Type |-> "aborted"] \
                              in messages
                          /\ temp' = temp \ {r}
                    /\ pc' = [pc EXCEPT ![coord] = "Lbl_6"]
               ELSE /\ pc' = [pc EXCEPT ![coord] = "Done"]
                    /\ temp' = temp
         /\ UNCHANGED << participants, messages, aborted >>

Lbl_7 == /\ pc[coord] = "Lbl_7"
         /\ IF temp /= {}
               THEN /\ \E r \in temp:
```

```
                              /\ messages' = (messages \union {[To |-> r, From |->
                                   coord, Type |-> "commit"]})
                              /\ temp' = temp \ {r}
                         /\ pc' = [pc EXCEPT ![coord] = "Lbl_7"]
                    ELSE /\ temp' = participants
                         /\ pc' = [pc EXCEPT ![coord] = "Lbl_8"]
                         /\ UNCHANGED messages
          /\ UNCHANGED << participants, aborted >>

Lbl_8 == /\ pc[coord] = "Lbl_8"
          /\ IF temp /= {}
                THEN /\ \E r \in temp:
                              /\ [To |-> coord, From |-> r, Type |-> "committed"]
                                  \in messages
                              /\ temp' = temp \ {r}
                         /\ pc' = [pc EXCEPT ![coord] = "Lbl_8"]
                ELSE /\ pc' = [pc EXCEPT ![coord] = "Done"]
                         /\ temp' = temp
          /\ UNCHANGED << participants, messages, aborted >>

Coordinator == Lbl_3 \/ Lbl_4 \/ Lbl_5 \/ Lbl_6 \/ Lbl_7 \/ Lbl_8

(* Allow infinite stuttering to prevent deadlock on termination. *)
Terminating == /\ \A self \in ProcSet: pc[self] = "Done"
               /\ UNCHANGED vars

Next == Coordinator
           \/ (\E self \in participants: P(self))
           \/ Terminating

Spec == Init /\ [][Next]_vars

Termination == <>(\A self \in ProcSet: pc[self] = "Done")

\* END TRANSLATION

=================================================================
```