Distributed systems are critical to reliable and scalable computing; however, they are complicated in nature and prone to bugs. To manage this complexity, network middleware has been traditionally built in layered stacks of components. We present a novel approach to compositional verification of distributed stacks to verify each component based on only the specification of lower components. We present TLC (Temporal Logic of Components), a novel temporal program logic that offers intuitive inference rules for verification of both safety and liveness properties of functional implementations of distributed components. To support compositional reasoning, we define a novel transformation on the assertion language that lowers the specification of a component to be used as a subcomponent. We prove the soundness of TLC and the lowering transformation with respect to a novel operational semantics for stacks of composed components in partially synchronous networks. We successfully apply TLC to compose and verify a stack of fundamental distributed components.

CCS Concepts:
• Theory of computation → Program analysis;
• Software and its engineering → Distributed programming languages;
• Distributed systems organizing principles.

Additional Key Words and Phrases: Distributed Protocols, Temporal Logic, Program Logic, Composition, Operational Semantics

ACM Reference Format:

1 INTRODUCTION

Distributed systems are the backbone of the modern computing infrastructure. They support the reliable, scalable and responsive execution of Internet services and replicated aviation control systems, and are at the core of crypto-currencies. However, due to their combinatorially large state spaces, and node and network failures, distributed systems are complicated and prone to bugs. Therefore, they repeatedly suffer data and currency loss, and service outage [Guo et al. 2013; Web 2018a,b,c]. Several projects [Dragoi et al. 2016; Hawblitzel et al. 2015; Lesani et al. 2016; Padon et al. 2016; Rahli 2012; Sergey et al. 2017; Wilcox et al. 2015] have been recently successful in verification of various distributed systems. However, they either do not benefit from a program logic and carry out verification in the semantic domain [Hawblitzel et al. 2015; Lesani et al. 2016; Wilcox et al. 2015], do not consider compositional reasoning [Dragoi et al. 2016; Hawblitzel et al. 2015; Padon et al. 2016; Rahli 2012], or do not verify liveness properties [Sergey et al. 2017].
Both operating systems, and and network middleware have been traditionally built in layers [Biagioni et al. 2001; Gu et al. 2016; Peterson and Davie 2003]. Each node hosts a stack of protocol layers and communicates with other nodes by the communication primitives at the bottom layers. This modular approach brings separation of the implementation from the interface. A higher layer only uses the interface and is separate from the implementation of the lower layers. Similarly, modular verification of each layer using only the specification of the lower layers reduces the proof engineering effort and brings scalability to the development of reliable distributed systems. Layers can be verified separately and composed to build verified stacks of distributed systems. Further, a layer remains correct if one of its lower layers is replaced with a new layer with the same specification.

This paper presents a novel framework for compositional specification and verification of distributed system stacks. Protocol designers and practitioners do reason about their distributed systems. We observe that they often state the properties of a protocol as natural language statements on events, assume the properties of the sub-protocols, and argue about correctness using intuitive arguments about the temporal precedence of the events that the protocol and the sub-protocols exchange [Cachin et al. 2011]. They find it more natural to state properties about the past events rather than add ghost state. Similarly, they prove liveness properties by simple reasoning about future events. This observation led us to the following questions: Can we capture the properties in a temporal logic for composable components? Can we capture the use of the specifications of subcomponents as a sound transformation? Can we formalize the principles used in these intuitive proofs as logical inference rules? Program logics have been traditionally developed as extensions of the classical Floyd-Hoare logic [Hoare 1969]. In the past decade, the community has witnessed increasingly complicated Hoare logics for concurrent programs that can be effectively used by experts. This project takes a distinct approach and strives to keep the formal techniques as close as possible to the practitioner language. It presents a new compositional and temporal approach to verification of stacks of distributed protocols.

We present a layered programming model to separately capture functional implementations of distributed components. Layers of components communicate through the interface of request and indication events: request events are input from the higher layer and output to the lower layers while indication events are input from the lower layers and output to the higher layer. We present a temporal assertion language on event traces to specify properties of components. The assertion language can naturally capture both safety and liveness properties of components in terms of their interface: incoming requests and outgoing indications. We present a novel program logic called TLC (Temporal Logic of Components) that features intuitive inference rules to directly reason about implementations of distributed components. We want to compositionally verify each component based on only its own implementation and the specifications of its subcomponents. Thus, we present a novel syntactic transformation to lower the temporal specifications of a component to be used as a subcomponent. We present an operational semantics for stacks of distributed components where events propagate across layers in a node, and nodes communicate via the bottom link layer in partially synchronous networks where nodes may crash. We prove the soundness of TLC and the lowering transformation with respect to the operational semantics. We successfully applied the programming model, the lowering transformation and TLC to compose and verify stacks of fundamental distributed components including stubborn links, perfect links, best-effort broadcast, uniform reliable broadcast, epoch Paxos consensus. Further, we present our progress in proof mechanization towards building certified middleware.

The main contribution of this paper is the novel program logic TLC and the lowering transformation for compositional verification of both safety and liveness properties of distributed components. On the whole, the paper makes the following contributions: (1) a compositional programming
model for distributed stacks (§ 2) and its novel semantics (§ 6), (2) a temporal assertion language on event traces to specify safety and liveness properties of components (§ 3), and a sound composable verification technique based on lowering specifications (§ 4), (3) a program logic to reason about components (§ 5) and its soundness with respect to the semantics (§ 7) and (4) verification of stacks of fundamental distributed components (the appendix [Appendix 2020] § 5.2), and the encodings and mechanized proofs in Coq (§ 8). We start with an overview.

2 OVERVIEW

In this section, we present compositional verification of a small property of a simple component that uses a subcomponent. We summarize the future sections of the paper including the necessary parts of the assertion language and selected inference rules. We present the specification of the properties of the component and the subcomponent, lower the specification of the subcomponent and then apply TLC to verify a property of the component. We present a small intuitive proof in the natural language in one paragraph and then illustrate how each step of this proof is directly supported by TLC.

Component Composition. We define the type of components Comp as a parametric record that is represented in Fig. 1. A component is parametric for the type of the events at the top and the bottom of the component as depicted in Fig. 2.(a). The events at the top are the interface of the component. They are the input requests of type IR and the output indications of type OI. A component may have multiple subcomponents. The events at the bottom are the output requests of types OR to and the input indications of types II from the subcomponents. We use the overline notation to denote multiple instances; for example, we use $\overline{OR}$ to denote multiple output request types, one per subcomponent. A component defines the State type and its initial value per node as the function init. It also defines three handler functions, request, indication and periodic, that are called in response to input request, input indication and periodic events. Periodic events are automatically issued regularly on correct nodes. Nodes may have crash-stop failures. A node is correct if it does not crash. The periodic handlers usually react to certain conditions; for example, when enough acknowledgements are received, an output indication is issued. Each of the three functions get the current node identifier (of type $\mathbb{N}$) and the pre-state of the component (of type State) as parameters. As the next parameter, the request function gets the input request from the higher
component and the indication function gets the input indication from one of the subcomponents. The handler functions return the post-state, a list of output requests (to subcomponents) and a list of output indications (to the parent component).

As Fig. 1 presents, we define the stack of components Stack as an inductive type that is parametrized on the interface of the stack. The interface of a stack is the top events IR and OI of its top component. A stack is inductively constructed as either a component and its matching substacks (as the inductive case) or a bottom link (as the base case). Basic links are the weakest components at the leaves of a stack. They accept requests send(n,m) of type Req, to send message m to node n and issue indications deliver(n,m) of type Ind to deliver message m from node n. The semantics of a link can drop messages. However, it does not unfairly drop a particular message that is repeatedly sent. If a sender keeps resending a message and the receiver has not failed, the message is eventually delivered.

As Fig. 2.(b) and (c) show, increasingly stronger components can be built on top of basic links: stubborn links, perfect links, best-effort broadcast, uniform reliable broadcast, epoch consensus, epoch change, and Paxos (uniform) consensus. We have spent extensive effort to write proofs of correctness for these components. The implementation, properties and detailed proofs of all these components are available in the appendix [Appendix 2020] § 4 and 5.3. Fig. 2.(b) shows the stack of the uniform reliable broadcast. Two identical stacks are drawn to show replication at two different nodes. The bottom horizontal lines show the low-level message passing by the basic link. Fig. 2.(c) shows the Paxos consensus [Lamport 1998] stack. The Paxos consensus component is at the top and uses epoch change and epoch consensus as its two subcomponents. In the epoch consensus component, a leader tries to impose a value to the correct nodes. The epoch change component initiates the next epoch with a new leader if the current one fails. The two subcomponents are horizontally composed and Paxos consensus is vertically composed on top of them.

The stubborn link repeatedly resends messages by the basic link so that they are eventually delivered. However, retransmission results in multiple deliveries that may not be desired by the higher-level components. Thus, the perfect link component is built on top of the stubborn link to eliminate duplicate messages. It keeps track of delivered messages and ignores duplicates. Fig. 3.(a) presents the perfect link component, PLC. (We will visit the parts (b) and (c) of Fig. 3 later in this section.) PLC provides the perfect link interface and uses a substack with the stubborn link interface. The state of each node stores the number of messages sent by the current node, counter, initialized to zero and the set of received message identifiers, received, initialized to empty (at $L_2-L_4$). The counter is used to assign unique numbers to messages that the node sends. Each message can be uniquely identified by the pair of the sender node identifier and the number of the message in that node. Upon a request to send a message (at $L_6-L_9$), the counter is incremented (at $L_{10}$) and the

```plaintext
Comp IR OI (OR,II) :=
(State: Type,
init: N → State,
let Out = State × List (Σ OR) × List OI in
request: N × State × IR → Out,
indication: N × State × Σ II → Out,
periodic: N × State → Out)

Stack: Type → Type → Type :=
| stack: Comp IR OI (OR,II) ×
Π (Stack OR II) →
Stack IR OI
| link: Stack Req Ind

Fig. 1. Component and Stack. IR: input requests type, OI: output indications type, OR: output requests types (one per subcomponent), II: input indications types (one per subcomponent). Σ and Π are parametric sum and product types. A component is defined as record of its State type, initialization function init and the three handler functions, request, indication and periodic, that are executed in response to input request, indication and periodic events. A stack is inductively constructed as either a component and its matching substacks, or a bottom link.
```
LCL 123:5

and indication events are shown as of a trace in Fig. 4. Components are represented as boxes and the orientation o
networks and node failures. Here, we illustrate the structure of a stack and a fragment of a round
the propagation of events across the stack, message passing across nodes in partially synchronous
descriptions, we write a sum term constructed from a term
if the message is already received, it is ignored (at
message is sent together with the new counter value using the stubborn link subcomponent (at
L11-L12). Upon a delivery indication of a message from the stubborn link subcomponent (at L15-L18),
if the message is already received, it is ignored (at L19-L20). Otherwise, the message identifier is
added to the received set and a delivery indication event is issued (at L21-L24). (In component
descriptions, we write a sum term constructed from a term t of the i-th type parameter as (i, t) for
brevity.) Finally, PLC does not need a periodic handler (at L27).

Semantics. In § 6, we define the operational semantics of distributed components. It models
the propagation of events across the stack, message passing across nodes in partially synchronous
networks and node failures. Here, we illustrate the structure of a stack and a fragment of a round
of a trace in Fig. 4. Components are represented as boxes and the orientation o of request, periodic
and indication events are shown as ↓, ↓ and ↑ respectively. Incoming events are executed on the
component itself and outgoing events are issued to be executed on other components. The distinct

location identifier $d$ of a component in the tree of a distributed stack is the reverse list of branch indices from the top component to that component. Going down and up the tree simply corresponds to adding and removing a subcomponent index at the head of this list. For example, the identifier $d$ for the top component $C_1$ is $\emptyset$, for its left child $C_2$ is $[0]$ and for its right grandchild $C_3$ is $[1,0]$. The interface of each component is the events immediately above it and they share its identifier. For example, the identifier $d$ of the right child $C_3$ and its interface events are both $[1]$. Similarly, the location identifier of a substack is the location identifier of its top component. For example, the left substack rooted at $C_2$ is at location $[0]$. A simple trace is shown on the right of Fig. 4 where the lines show a sequence of events from left to right at different interface levels. The execution of an event at a component updates the state of the component and may issue other request and indication events. The issued events are subsequently executed. The trace starts with $e_1$, a request $\downarrow$ at the top $\emptyset$ from the client. When $e_1$ is processed in the top component $C_1$ at $\emptyset$, $e_2$, a request $\downarrow$ on the left child component $C_2$ at $[0]$ is issued. When $e_2$ is processed in $C_2$, in turn, $e_3$, a request $\downarrow$ to its right child $C_5$ at $[1,0]$, is issued. Processing of $e_3$ on $C_5$ issues $e_4$, an indication $\uparrow$ event at $[1,0]$. When $e_4$ is processed in the parent component $C_2$ at $[0]$, $e_5$ that is an indication $\uparrow$ at $[0]$, is issued. (We note that $C_2$ could instead issue another request like $e_3$ to one of its children.) When $e_5$ is executed at the parent component $C_1$ at $\emptyset$, finally, $e_6$ that is an indication $\uparrow$ at the top level $\emptyset$, is issued (that is executed in the client). An (infinite) sequence of event labels is an execution trace. Given a stack, the semantics defines its set of execution traces.

**Assertion Language.** To represent the specifications of distributed component stacks, we define a temporal assertion language that can describe traces of events across the stack. It features specific variables for the properties of the handler calls and a location variable to distinguish the unique places of the composed components in the stack. It can concisely capture safety and liveness properties of distributed components. In §3, we will describe the assertion language and here, briefly describe the parts that we use in the overview. The always assertion $\square A$ states that the assertion $A$ holds at every event in the future including the current event. The always in the past assertion $\boxdot A$ states that $A$ holds at every event in the past including the current event. The eventually assertion $\Diamond A$ states that $A$ holds at some future event. The eventually in the past assertion $\diamondsuit A$ states that $A$ holds at some past event. The strict versions $\widehat{\Diamond}$, $\widehat{\Box}$, $\diamondsuit$ and $\boxdot$ exclude the current event. The strong implication $A \Rightarrow A'$ is syntactic sugar for $\square(A \rightarrow A')$ where $\rightarrow$ is the logical implication. The leads-to assertion $A \rightsquigarrow A'$ is syntactic sugar for $\square(A \rightarrow \Diamond A')$. Similarly, the preceded-by assertion $A \leftsquigarrow A'$ is syntactic sugar for $\square(A \rightarrow \Box A')$. An assertion is non-temporal if it does not include any temporal operators.

The user events are the event objects that the protocol handlers take as argument and issue, for example $\text{send}_m((n,m))$. A trace event represents the execution of a user event by a handler. The assertion language can describe event traces across the stack. Variables are partitioned into rigid and flexible variables. A rigid variable has the same value in all events of an execution, while a flexible variable may assume different values in different events. We represent the flexible variables with the bold face. The flexible variables for an event are the identifier $n$ of the node that executes the event, the round number $r$ that executes the event, location identifier $d$ that the event is executed at, the orientation $o$ of the event, the user event $e$ that is processed, the output requests $\textbf{ors}$ and output
indications ois that the event issues, the pre-state s of the event, and the post-state s' of the event. The pre and the post-state represent functions from node identifiers to the state of the component at the nodes. We call the top component self as it is the top component that is verified assuming the correctness of the subcomponents. We use the syntactic sugar assertion self to describe events that are applied to the top component. A self event is either a request or periodic event at the top or an indication event from a subcomponent at the second level. The constants Correct represents the set of identifiers of correct node, i.e., the nodes that have not crashed.

The syntactic sugar assertion n • A (where • is used as a separator) is sugar for n = n ∧ A; it describes an event that is executed at node n and satisfies A. The syntactic sugar assertion ⊤ o e stands for d = [] ∧ o = o ∧ e = e; it describes an event that is at the top (⊤) level interface [], its orientation is o (either the constant ↓ for requests, ↓ for periodic or ↑ for indications) and its user event is e. For example, the assertion (n • ⊤ ↓ send3i(n',m)) describes an event at node n at the top level interface [] where the request (↓) event send3i(n',m) is executed. As the periodic handler is not called with a user event, we use the constant per to represent periodic user events. Similarly, the syntactic sugar assertion i o e stands for d = [i] ∧ o = o ∧ e = e; it describes an event that is at the interface of the i-th subcomponent at location [i], its orientation is o and its user event is e. For example, the assertion (n • 1 ↑ deliver3i(n',m)) describes an event at node n at the interface location [1] where the indication (↑) event deliver3i(n',m) is executed. It is notable that as a pleasant result of compositional reasoning, we only need to refer to the events at the top and events at the second level. Therefore, we defined syntactic sugar for only the first two levels.

Specifications. Fig. 3.(b) and (c) shows the specification of stubborn links and perfect links that are written almost verbatim from their natural language descriptions. A stubborn link stubbornly retransmits messages. The stubborn delivery property SL1 states that once a message is sent, it is delivered infinitely often. The no-forge property SL2 states that a stubborn link never forges a message. The properties SL1 and SL2 are liveness and safety properties respectively. Intuitively, a safety property states that a bad state never happens and a liveness property states that a good state eventually happens. The reliable delivery property PL1 states that perfect links can reliably transmit messages between correct nodes. The no-duplication property PL2 states that perfect links do not redundantly deliver messages. (It is notable that (p ⇒ ˆ□p) → (p ⇒ ˆ□p); hence the ˆ□ conjunct is omitted in the no-duplication property.) The no-forge property PL3 states that perfect links do not forge messages. The property PL1 is a liveness and the properties PL2 and PL3 are safety properties.

The specification of a component is simply written when it is the main component at the top of the stack in terms of its interface: its incoming requests and outgoing indications. It can then be lowered to any subcomponent location. We note that as the location is always the top ⊤ in specifications, it can be elided in specifications for brevity. Further, if we add the convention that the event names such as send and deliver are used for only either requests or indications, the orientation arrows ↓ for request and ↑ for indication can be removed from the specifications as well; they can be derived from the event. For example, send is a always a request and deliver is always an indication. Then, for example, the no-forge property of stubborn links SL2 in Fig. 3 can be summarized as the following assertion: (n • deliver3i(n',m)) ↔ (n' • send3i(n,m)). If a node n delivers a message m with sender n', then n' previously sent m to n. This specification seem to be the bare minimum to match the corresponding natural language statement. To keep the uniformity of assertions, we make the location and orientation explicit in the specifications.

Lowering Specifications. We now showcase lowering specifications and the program logic inference rules with the short proof of the no-forge property of the perfect link component PLC.

PLC uses the stubborn link interface and relies on its properties. The specification of the stubborn link in Fig. 3.(b) is stated on its interface as the top-level component but PLC uses it as a
Fig. 5. Illustration of Lowering. (a) Stack $S_0$ and its specification $I_0$. (b) Stack $S_1$ and its specification $I_1$. (c) Stack $S$ that composes $S_0$ and $S_1$ as subcomponents. To prove the assertion $A$ for $S$, it is sound to assume the lowering of $I_0$ and $I_1$, and derive $A$ in TLC, i.e., $\text{lower}(i, I_i) \vdash c A$.

subcomponent. When a component is used as a subcomponent, its events are at lower locations and are also interleaved with the events of the parent and sibling components. Given the top-level specification of the stubborn link, how can we transform it to be used as the specification of a subcomponent for the perfect link? Not every assertion can be lowered. In § 4, we present an invariant assertion language that is restrictive enough to be lowered by a syntactic transformation and expressive enough to capture component specifications.

Fig. 5 illustrates the lowering transformation lower. The specification of each stack $S_i$ is given as an invariant $I_i$ (Fig. 5.(a) and (b)). Consider that we have a stack $S$ with the component $c$ at the top and the substacks $S_i$, i.e., $S = \text{stack}(c, S_i)$. What can we assume for each subcomponents $S_i$? We define the translation function $\text{lower}$ on invariants and show that to prove the validity of $A$ for $S$, it is sound to assume $\text{lower}(i, I_i)$ and derive $A$ in TLC, i.e., $\text{lower}(i, I_i) \vdash c A$.

The SL assertion is in the invariant sub-language. Applying the lowering transformation to SL to use it as the 0-th subcomponent results in the following:

$$\text{SL}'_2 = \text{lower}(0, \text{SL}_2) = \text{lower}(0, (n \bullet \top \uparrow \text{deliver}_{sl}(n', m)) \leftrightarrow (n' \bullet \top \downarrow \text{send}_{sl}(n, m))) = (n \bullet 0 \uparrow \text{deliver}_{sl}(n', m)) \leftrightarrow (n' \bullet 0 \downarrow \text{send}_{sl}(n, m))$$

We will see the transformation details later in § 4, but notice here that top $\top$ is changed to 0. The lowering transformation can be similarly applied to SL to result in SL$_1'$.

The judgements of the logic are of the form $\Gamma \vdash c A$ that states that under assumptions $\Gamma$, the assertion $A$ holds for the component $c$. We assume the two lowered assertions; thus, we have $\Gamma = \text{SL}_1', \text{SL}_2'$.

**Program Logic.** We now showcase the program logic using a simple example. The no-forge property of perfect links states that a perfect link delivery event is preceded by a corresponding perfect link send event. We want to apply TLC to prove the following judgement that states that assuming $\Gamma$, the no-forge property is valid for PLC.

$$\Gamma \vdash_{\text{PLC}} (n \bullet \top \uparrow \text{deliver}_{pl}(n', m)) \leftrightarrow (n' \bullet \top \downarrow \text{send}_{pl}(n, m))$$

At a high-level level, the proof shows a precedence sequence that transitively imply the desired precedence. The proof steps are illustrated in Fig. 7. Step 1: A perfect link delivery event is executed; hence, the event should have been previously issued. Step 2: By the component implementation
(in Fig. 3(a)), a perfect link delivery event is issued by only the indication handler function. Thus, the issuing event is a stubborn link delivery event. Step 3: By the no-forge property of stubborn links, a stubborn link delivery is preceded by a stubborn link send. Step 4: A stubborn link send event is executed before; thus, it should have been previously issued. Step 5: By the component implementation (in Fig. 3(a)), a stubborn link send event is issued by only the request handler function. Thus, the functional implementation of the component can be directly concluded form the above steps that a perfect link delivery event is preceded by a perfect link send event.

The rules and lemmas that we use for this proof are presented in Fig. 6. We will look at the rules closely in § 5. Here, we use two basic rules: rule OR' and rule OI', one derived rule: rule Invl and two basic temporal logic lemmas Trans and TransΩ. Intuitively, the two rules OR' and OI' state that if an event is executed, it should have been previously issued. The rule OR' states that if at a node n and the subcomponent i, a request \(i \downarrow e\) event e is processed, then in the past, at the same node n, the request \((i,e)\) is issued by a self event. Similarly, the rule OI' states that if at a node n and at the top level \(\top\), an output indication \(\top\) event e is processed, then in the past, at the same node n, the indication e is issued by a self event. The rule Invl states that if a non-temporal assertion holds for all the three handler functions of the component, request, periodic and indication, then the assertion holds in every self event. It is notable that Invl reduces a temporal global assertion to non-temporal local proof obligations: each premise of this rule is a non-temporal assertion about a single handler function. Thus, the functional implementation of the component can be directly used to infer its properties. The two temporal logic lemmas Trans and TransΩ state basic temporal transitivity properties. By rule OI', we have

\[
\text{Step 1: } \Gamma \vdash_{\text{PLC}} (n \bullet \top \downarrow \text{deliver}_{\pi}(n',m)) \Rightarrow \Diamond (n \bullet \text{deliver}_{\pi}(n',m) \in \text{ois} \land \text{self})
\]

that states that if a perfect link indication is executed, it is previously issued by a self event. We now prove that it is issued only when a stubborn link delivery is executed. We use rule Invl with

\[
\text{Proc. ACM Program. Lang., Vol. 4, No. ICFP, Article 123. Publication date: August 2020.}
\]
Considering the implementation in Fig. 3.(a), the two cases for request and periodic are straightforward as \( \text{ois} = [] \) in both and the premise is refuted. The other case is for indication where the stubborn link indication is executed. Thus, by rule InvL (and then reducing two implications to one), we have:

\[
\text{Step 2: } \Gamma \vdash_{\text{PLC}} (\text{self} \land n \bullet \text{deliver}_{pl}(n', m) \in \text{ois}) \Rightarrow \exists c. (n \bullet 0 \uparrow \text{deliver}_{sl}(n', \langle c, m \rangle)) \tag{4}
\]

By Lemma Trans\(\otimes\) on Eq. 2 and Eq. 4, and existential elimination for \(c\), we have

\[
\Gamma \vdash_{\text{PLC}} (n \bullet \top \uparrow \text{deliver}_{pl}(n', m)) \Rightarrow \Theta(n \bullet 0 \uparrow \text{deliver}_{sl}(n', \langle c, m \rangle)) \tag{5}
\]

that states that the perfect link delivery event is preceded by a stubborn link delivery event.

From \(\Gamma\), and Eq. 1 (lowered \(\text{SL}_2\)), instantiating \(m\) with \(\langle c, m \rangle\) and unfolding \(\otimes\), we have

\[
\text{Step 3: } \Gamma \vdash_{\text{PLC}} (n \bullet 0 \uparrow \text{deliver}_{sl}(n', \langle c, m \rangle)) \Rightarrow \Theta(n' \bullet 0 \downarrow \text{send}_{sl}(n, \langle c, m \rangle)) \tag{6}
\]

that is the assumption that every stubborn link delivery event is preceded by a stubborn link send event. By rule OR\(\prime\), we have

\[
\Gamma \vdash_{\text{PLC}} (n' \bullet 0 \downarrow \text{send}_{sl}(n, \langle c, m \rangle)) \Rightarrow \Theta(n' \bullet (0, \text{send}_{sl}(n, \langle c, m \rangle)) \in \text{ors} \land \text{self}) \tag{7}
\]

that states that every executed stubborn link send event is previously issued by a self event. We use rule InvL again with the assertion

\[
A = n' \bullet (0, \text{send}_{sl}(n, \langle c, m \rangle)) \in \text{ors} \rightarrow (n' \bullet \top \downarrow \text{send}_{pl}(n, m))
\]

Considering the implementation in Fig. 3.(a), the two cases indication and periodic are straightforward as \(\text{ors} = []\) in both. The other case is request where the perfect link request is executed. Thus, we have

\[
\text{Step 5: } \Gamma \vdash_{\text{PLC}} (\text{self} \land n' \bullet (0, \text{send}_{sl}(n, \langle c, m \rangle)) \in \text{ors}) \Rightarrow (n' \bullet \top \downarrow \text{send}_{pl}(n, m)) \tag{8}
\]

By Lemma Trans\(\otimes\) on Eq. 7 and Eq. 8, we have

\[
\Gamma \vdash_{\text{PLC}} (n' \bullet 0 \downarrow \text{send}_{sl}(n, \langle c, m \rangle)) \Rightarrow \Theta(n' \bullet \top \downarrow \text{send}_{pl}(n, m)) \tag{9}
\]

From Lemma Trans\(\otimes\otimes\) on Eq. 5, Eq. 6 and Eq. 9, we have

\[
\Gamma \vdash_{\text{PLC}} (n \bullet \top \uparrow \text{deliver}_{pl}(n', m)) \Rightarrow \Theta(n' \bullet \top \downarrow \text{send}_{pl}(n, m))
\]

that is

\[
\Gamma \vdash_{\text{PLC}} (n \bullet \top \uparrow \text{deliver}_{pl}(n', m)) \leftrightarrow (n' \bullet \top \downarrow \text{send}_{pl}(n, m))
\]

The implementations, specifications and proofs of the other components are available in the appendix [Appendix 2020] § 4 and 5.2. After this overview, we first define the assertion language (§ 3). Next, we define the lowering transformation and prove its soundness for compositional reasoning (§ 4). Then, we present TLC inference rules (§ 5). We finally present the mechanization framework (§ 8).

### 3 ASSERTION LANGUAGE

We now present the assertion language of TLC in Fig. 8. It is a temporal language on event traces of stacks composed of distributed components. It can concisely capture both safety and liveness properties. We have already seen parts of the language in § 2; we consider the rest here.

The constants \(\mathbb{N}\) and Correct are the set of participating and correct node identifiers respectively. Similar to classical first-order logic, a term \(t\) can be a variable, a constant or an application of a function \(f\) to other terms. As the pre-state \(s\) and post-state \(s'\) variables have function values, a term can be constructed by applying a variable to terms as well.

An atomic assertion is an application of a predicate \(p\) to terms. All propositional and quantified formula can be constructed as syntactic sugar to conjunction, negation and universal quantification.
when it is the main component at the top of the stack; however, it should be later used as the main component of the stack. The goal of compositional verification is to verify the component using only the specifications (and not the implementations) of the subcomponents. The specification of each component is written when it is at the top of the stack. A fundamental question is how the specification of a component is written.

4 SPECIFICATION LOWERING

The presented programming model allows a component to be programmed using subcomponents. The goal of compositional verification is to verify the component using only the specifications (and not the implementations) of the subcomponents. The specification of each component is written when it is the main component at the top of the stack; however, it should be later used as the specification of a subcomponent.

Assertion for the stack at location $d$:

$$ A^d := A \text{ such that } a := (n = t \land d = d' \land o = t \land e = t) \land d' \supseteq d $$

Invariant for the stack at location $d$:

$$ \mathcal{I}^d := \Box A^d $$

Invariant:

$$ \mathcal{I} := \Box A \text{ such that } \circ \text{ and } \circ \text{ are not used.} $$

Syntactic Sugar:

$$ n \cdot A \triangleq n = n \land A $$

$$ \top \circ e \triangleq e = e $$

$$ i \circ e \triangleq d = \{i\} \land o = o \land e = e $$

$$ \text{self} \triangleq (d = \{\} \land o = \{\}) \lor (\exists i. d = \{i\} \land o = \{\}) $$

Similarly, the grammar only shows the strict versions of the temporal operators as the non-strict versions can be defined as syntactic sugar. The temporal operators that we did not introduce in §2 are $\circ$ and $\circ$. The next operator $\circ$ states that its operand assertion holds in the immediate next event. The self subtrace is the sequence of events executed on the top component. The self operator $\circ$ allows stating assertions about the self subtrace. The assertion $\circ A$ asserts $A$ on the self subtrace. The self operator is usually used as the outermost operator.

We use $\mathcal{I}^d$ to represent invariant assertions for the substack at location $d$. The specification of a component is written when it is at the top $\{\}$ as a top-level invariant $\mathcal{I}$. An invariant $\mathcal{I}^d$ is of the form $\Box A^d$. To support lowering, the atomic assertions used in an assertion $A^d$ are constrained to have the form $n = t \land d = d' \land o = i' \land e = e$ such that $d' \supseteq d$. The location variable $d$ is explicitly equal to an extension $d'$ of $d$, i.e., the event is executed under the substack at location $d$. For example, the event at location $[2,1,0]$ is executed under the $0$-th child of the top component, i.e., $[2,1,0] \supseteq [0]$. The assertion also includes explicit equalities for the executing node $n$, the orientation $o$ and the user-level event $e$. Further, invariant assertions do not use the next $\circ$ or self $\circ$ operators.
should be lowered to be used as a subcomponent. The lowered specifications of the subcomponents are used as assumptions to verify the specification of the new parent component (that is programmed on top of the subcomponents). In this section, we define the lowering transformation on specification assertions and prove its soundness. Lowering is not possible for every assertion. We observed that lowering specifications requires certain information, such as the location of events, to be present and certain operators, such as next, to be absent from the specification. We identify a subset of the assertion language that is both restrictive enough to allow the definition of the lowering transformation and expressive enough to represent specifications.

As we saw in Fig. 5, the specification of each stack $S_i$ is given as an invariant $\overline{I}_i$ (Fig. 5(a) and (b)). We have a stack $S$ with the component $c$ at the top and the substacks $S_i$, i.e., $S = \text{stack}(c, S_i)$. We want to verify that $S$ satisfies its specification $A$ (Fig. 5(c)). We define the translation function lower on invariants and show that to prove the validity of $A$ for $S$, it is sufficient to assume $\text{lower}(i, \overline{I}_i)$ and derive $A$ in TLC. Fig. 9 represents the function lower on the invariant sub-language $\overline{I}_i$. It first pushes and then restricts the assertion. We visit each in turn.

**Pushing.** For the component at the top, the semantics of stacks models the most general client that may issue any request. However, when the component is used as a subcomponent, the parent component may only issue a subset of the possible requests. Therefore, if a stack is pushed from the top to a lower layer, its set of subtraces can only become smaller (or stay the same). Thus, the specification of a stack at the top level can serve as a starting point for its specification as the substack $i$. However, the stack is now used at a deeper level. The location of every event of the stack is now under branch $i$. For example, the location of its highest events is $[]$ when it is at the top and is $[i]$ when it is at the $i$-th substack. Similarly, an event at location $[1]$ is pushed to location $[1, i]$. Therefore, the first transformation is to push the locations under branch $i$. The function push is defined in Fig. 9. As we saw in the definition of the invariant sub-language $\overline{I}_i := \square A_i$, the location values $d'$ are explicit in assertions $A_i$. Given a top-level assertion $A_i$ and a branch index $i$, the function push translates the location value from $d'$ to $i :: d'$. Appending $i$ to $d'$ effectively pushes the events to branch $i$.

---

**Definition 1 (Lowering Assertions).** \(\text{lower}(i, \overline{I}_i) \triangleq \text{restrict}(d \geq [i], \text{push}(i, \overline{I}_i))\)

**Definition 2 (Pushing an Assertion).**

\[
\begin{align*}
\text{push}(i, A_1) & : \neg push(i, A_1) & \text{Definition 3 (Restricting an Assertion).} \\
\text{push}(i, \forall x. A_1) & : \forall x. \text{push}(i, A_1) \\
\text{push}(i, \exists x. A_1) & : \exists x. \text{push}(i, A_1) \\
\text{push}(i, \Diamond A_1) & : \Diamond \text{push}(i, A_1) \\
\text{push}(i, \Diamond^* A_1) & : \Diamond^* \text{push}(i, A_1)
\end{align*}
\]

Fig. 9. Lowering (Pushing and Restricting) Assertions. An atomic assertion is denoted by $a$. 

---

\[
SL'_2 = \text{lower}(0, SL_2) =
\]
\[
\text{lower}(0, (n \cdot \top \uparrow \text{deliver}_{st}(n', m)) \iff (n' \cdot \top \downarrow \text{send}_{st}(n, m)))
\]
\[
\text{restrict}(d \supseteq [0], \Box[(n \cdot \top \uparrow \text{deliver}_{st}(n', m)) \rightarrow (n' \cdot \top \downarrow \text{send}_{st}(n, m))]) =
\]
\[
\text{restrict}(d \supseteq [0], (n \cdot 0 \top \uparrow \text{deliver}_{st}(n', m)) \rightarrow (n' \cdot 0 \downarrow \text{send}_{st}(n, m))]) =
\]
\[
\Box[(n \cdot 0 \top \uparrow \text{deliver}_{st}(n', m)) \rightarrow (n' \cdot 0 \downarrow \text{send}_{st}(n, m))]) =
\]
\[
(n \cdot 0 \top \uparrow \text{deliver}_{st}(n', m)) \iff (n' \cdot 0 \downarrow \text{send}_{st}(n, m))
\]

Fig. 10. Lowering Example

Restricting. When a stack is at the top, all events belong to that stack. However, when it is pushed to a substack, its events are interleaved with events from the top component and the sibling substacks. Therefore, the second transformation is to restrict the specification to remain valid on traces that are extended with interleaving events. Consider a specification \( \Box A \) for a stack. After pushing the assertion to the \( i \)-th substack, the resulting assertion \( \Box \text{push}(i, A) \) does not necessarily remain valid because although the assertion \( \text{push}(i, A) \) is valid on events under branch \( i \), it may simply not be valid on events of the top component and the sibling substacks. Thus, the restricting condition of being under branch \( i \) should be added and the assertion \( \Box \text{push}(i, A) \) is translated to \( \Box(d \supseteq [i] \rightarrow \text{push}(i, A)) \). (The assertion \( \text{push}(i, A) \) should be recursively translated as well.) As the definition of the function restrict in Fig. 9 shows, the other variants of the always operator are translated similarly. On the other hand, an eventually assertion \( \Diamond A \) remains the same. If an event will happen in the future, it will still happen if other events are interleaved before it. Similarly, the other variants of the eventually operator remain the same.

As an example, Fig. 10 elaborates lowering of SL2 that we saw in Eq. 1. The property SL2 is in the invariant language \( \mathcal{T}[1] \) and can be easily lowered by the syntactic transformation. The steps follow the definition in Fig. 9. We only explain a couple of subtleties. In Eq. 12 - Eq. 13, the push function translates \( d = [0] \) to \( d = [0] \). Thus, in the syntactic sugar, \( \top \) is translated to 0. In Eq. 14, the syntactic sugar \( (n \cdot 0 \top \uparrow \text{deliver}_{st}(n', m)) \) includes the conjunct \( d = [0] \). From basic propositional logic, for any \( A \) and \( A' \), we have that \( d \supseteq [0] \rightarrow ((d = [0] \land A) \rightarrow A') \) simplifies to \( (d \supseteq [0] \land d = [0] \land A) \rightarrow A' \). Since \( d = [0] \supseteq [0] \), it further simplifies to \( (d = [0] \land A) \rightarrow A' \).

We note that if the location was not explicit in the assertion, the assertion could not be pushed and would remain too general. For example, consider the assertion \( e = \text{send}(n, m) \Rightarrow m > 0 \) for a stack \( S_1 \) that states that all the messages that it sends are positive. This assertion is too general for a stack \( S \) that composes \( S_1 \) as a subcomponent because the top component or the sibling subcomponents may send negative messages. Further, if the assertions included the next operator \( O \), they could not be restricted to remain valid after the trace is interleaved with events of the top component and the sibling subcomponents. For example, consider the assertion \( (d = [] \land o = \downarrow) \Rightarrow O (d = [] \land o = \top) \) for a stack \( S_1 \) that states that every top-level request is immediately followed by a indication. The pushed assertion \( (d = [1] \land o = \downarrow) \Rightarrow O (d = [1] \land o = \top) \) is not valid for a stack \( S \) that composes \( S_1 \) as a subcomponent because the events of the other components may be interleaved between the request and indication. Similarly, assertions that use the self operator \( \circ \) cannot be lowered. For example, consider the assertion \( \circ (\forall n. s(n) > 0) \) for a stack \( S_1 \) that states that the state of the top component of \( S_1 \) remains positive. Obviously, this assertion does not necessarily hold for the new top component. Thus, the next and self operators are not useful for the specification of
components. However, they are useful for intermediate verification steps. For example, the basic inference rule PostPre that we will see in the next section uses both of these operators.

**Soundness.** The following theorem states the soundness of the lowering transformation for compositional reasoning. If a top-level invariant \( I_i \) is valid for the stack \( S \), and \( S \) is a substack of the stack \( T \), then the lowered invariant lower\((i, I_i)\) is valid for \( S \). We use the validity judgement \( \models_S A \) that states that \( A \) is valid in every trace of \( S \). (Validity is defined more precisely in § 7.)

Theorem 1. For all \( S, c, \) and \( S_i \), such that \( S = \text{stack}(c, S_i) \), if \( \models_{S_i} I_i \) then \( \models_S \text{lower}(i, I_i) \).

We now state the compositional proof technique and its soundness. The specifications of substacks can be lowered and used to derive the specification of the stack. Judgements of TLC are of the form \( \Gamma \vdash_c A \) where \( \Gamma \) is the assumed assertions and \( A \) is the deduced assertion. Consider valid top-level invariants \( I_i \) for stacks \( S_i \), and a stack \( S \) built by the component \( c \) on top of \( S_i \). The following theorem states that assuming the lowered invariants lower\((i, I_i)\), any assertion that TLC deduces for \( c \) is valid for \( S \).

Corollary 1 (Composition Soundness). For all \( S, c, \) and \( S_i \) such that \( S = \text{stack}(c, S_i) \), if \( \models_{S_i} I_i \) and lower\((i, I_i) \vdash_c A \) then \( \models_S A \).

5 TLC INFEERENCE RULES

In this section, we present the basic and derived inference rules of TLC. The basic inference rules are intuitive and fit in half a page. Yet, they provide the basis for verification of full stacks such as Fig. 2.(b) and (c). During the verification of the use-case protocols, we incrementally captured the fundamental reasoning steps as the basic rules. Further, we captured the higher-level reasoning steps as derived rules. The sequent judgements are of the form \( \Gamma \vdash_c A \) where \( \Gamma \) is the component, \( \Gamma \) is a set of assumed assertions and \( A \) is the deduced assertion. The inference rules axiomatize the properties of the semantics and the low-level communication primitive. More importantly, they allow deducing assertions about execution traces from the functional definition of the component. Fig. 11 presents the basic inference rules of TLC. (We elide the standard rules of sequent logic to the appendix [Appendix 2020] § 1.1). Further, we present a few derived rules in Fig. 12. A full list of derived rules are available in the appendix [Appendix 2020] § 1.3.

The first three rules IR, II and Pe state that when an event is executed on the top component, the corresponding handler function of the component is applied. These rules take the reasoning to the functional definition of the component. (Rule IR): The rule IR (for input request) states that if at the top level \( \top \), a request \( \downarrow \) event \( e \) is executed then the request \( \top \) handler function is called at that step. The request \( \top \) function of the component \( c \) represents a relation between its inputs: the stepping node \( n \), the pre-state \( s(n) \) of \( n \), and the user event \( e \), and its outputs: the post-state \( s'(n) \), the issued output requests \( \text{ors} \) and the issued output indications \( \text{ois} \). The rules II (for input indication) and Pe (for periodic) are similar. (Rule II): The rule II states that if an indication \( \top \) event \( e \) from the \( i \)-th subcomponent is executed then the indication \( \top \) handler function is called. (An indication event \( e \) from a subcomponent \( i \) is passed to the indication \( i \) function as the sum term \( (i, e, \top) \)) (Rule Pe): The rule Pe states that if at the top \( \top \), a periodic \( \xi \) event per is executed then the periodic \( \xi \) handler function is called.

The next four rules axiomatize the relation of issued and executed events. The rules OR and OI state that an event that is issued for a component is eventually executed on the component, and the rules OR' and OI' state that executed events are previously issued. These rules let the reasoning follow a chain of steps. (Rule OR): An output request from the top component is an
IR
\[\vdash c \top \downarrow e \Rightarrow (s'(n) \land \text{ors}, \text{ois}) = \text{request}_c(n, s(n), e)\]

II
\[\vdash_c i \uparrow e \Rightarrow (s'(n) \land \text{ors}, \text{ois}) = \text{indication}_c(n, s(n), (i,e))\]

PE
\[\vdash_c \top \not\in \text{per} \Rightarrow (s'(n) \land \text{ors}, \text{ois}) = \text{periodic}_c(n, s(n))\]

OR
\[\vdash_c n \cdot (i,e) \in \text{ors} \land \text{self} \Rightarrow \Diamond (n \cdot i \downarrow e)\]

OI
\[\vdash_c n \cdot (i,e) \in \text{ois} \land \text{self} \Rightarrow \Diamond (n \cdot \top \uparrow e)\]

APER
\[\vdash_c \text{Correct} \iff \square \Diamond (n \cdot \top \not\in \text{per})\]

ASelf
\[\vdash_c \text{init} \iff \square \Diamond (n \cdot \top \not\in \text{per})\]

SINV
\[\vdash_c (\text{init}) \iff \text{init}(n)\]

INIT
\[\vdash_c (s' = s \iff \square s = s)\]

SEQ
\[\vdash_c n \neq n' \Rightarrow s'(n) = s(n)\]

RSEQ
\[\vdash_c r = r \Rightarrow \Diamond (r \leq r)\]

GST
\[\vdash_c n' \in \text{Correct} \land \top \geq r_{\text{GST}} \land \Diamond (n \cdot d \downarrow \text{send}(n', m) \land r = r) \Rightarrow \Diamond (n' \cdot d \downarrow \text{send}(n', m))\]

FDUP
\[\vdash_c \square \Diamond (n' \cdot d \uparrow \text{deliver}(n, m)) \Rightarrow \Diamond (n \cdot d \downarrow \text{send}(n', m))\]

NForge
\[\vdash_c (n' \cdot d \uparrow \text{deliver}(n, m)) \Rightarrow \Diamond (n \cdot d \downarrow \text{send}(n', m))\]

NODE
\[\vdash_c \square n \in \mathbb{N}\]

UNITOR
\[\vdash_c (\text{occ}(\text{ors}, e) \leq 1 \land \square \Diamond (n = n \land \text{self} \rightarrow (i,e) \notin \text{ors}) \land \square \Diamond (n = n \land \text{self} \rightarrow (i,e) \notin \text{ors}) \Rightarrow (n \cdot i \downarrow e) \Rightarrow \square \neg (n \cdot i \downarrow e) \land \square \neg (n \cdot i \downarrow e)\]

Fig. 11. TLC Basic Inference Rules. We use \text{request}_c, \text{indication}_c and \text{periodic}_c to refer to the handler functions of the component $c$. The function call \text{occ}(l,e) counts the number of the element $e$ in the list $l$.

The event $e$ to a subcomponent $i$ that is issued as the sum term $(i,e)$. The rule OR (for output request) states that if at a node $n$, an output request $(i,e)$ is issued by a self event, then eventually at $n$ and the subcomponent $i$, the request $\downarrow$ event $e$ is executed. (Rule OI): Similarly, the rule OI (for output indication) states that if at a node $n$, an output indication $e$ is issued by a self event, then eventually at $n$ and the top level $\top$, the indication $\uparrow$ event $e$ is executed. The rules OR' and OI' state the relation of issued and executed events in the opposite direction of the rules OR and OI. (Rule OR'): The rule OR' states that if at a node $n$ and a subcomponent $i$, a request $\downarrow$ event $e$ is executed, then in the past, at that node $n$, the request event for that subcomponent $(i,e)$ is issued by a self event. (Rule OI'): Similarly, the rule OI' states that if at a node $n$ and the top level $\top$, an output indication $\uparrow$ event $e$ is executed, then in the past, at that node $n$, that indication event $e$ is issued by a self event. (Rule APER): The rule APER (for always periodic) states that if a node is correct, it infinitely often executes the periodic event.

(Rule ASelf): The self subtrace is the sequence of events executed on the top component. The rule ASelf (for always self) states that every event in the self subtrace is a self event. (Rule SINV): The rule SINV (for self invariant) states that an invariant for the self subtrace can be transformed to an invariant for the whole trace and vice versa using the restrict function that we saw earlier in Fig. 9. An invariant $\overline{I}$ holds in the self subtrace if and only if the restriction of the assertion restrict($self, \overline{I}$) holds in the whole trace. As the invariant continues to hold in the self events of the whole trace, the restriction condition is the self assertion. The rules ASelf and SINV lead to the derived rule INVLS (in Fig. 12). (Rule INVLS): The rule INVLS states that if a non-temporal assertion holds for all the three handler functions, request, indication and periodic, then the assertion holds in every
We note that this assertion does not hold on the whole trace as the events of different components

\[ \text{IR} \]

which is the same assertion as the rule \( \text{IR} \).

Therefore, the request event \( \text{request} \) (for the whole trace) to derive the same assertions under the self operator (for the self subtrace).

Then, for the three cases, the rule \( \text{SInv} \) is applied to the assertions that the rules \( \text{IR}, \text{II} \) and \( \text{Pe} \) state (for the whole trace) to derive the same assertions under the self operator (for the self subtrace).

For example, after basic logical rewriting, the assertion of the rule \( \text{IR} \) can be restated as

\[ \vdash_c \square_c (\exists e. (\top \downarrow e) \lor (s \uparrow e) \lor (\top \notin \text{per})) \]  \hspace{1cm} (17)

that is equivalent to the following assertion. (We remember that \( A \Rightarrow A' \) is defined as \( \square(A \Rightarrow A') \).)

\[ \vdash_c \text{restrict}(\text{self}, \top \downarrow e) \Rightarrow \text{request}_c(n, s(n), e) \]  \hspace{1cm} (19)

Then, the result of applying the rule \( \text{SInv} \) is

\[ \vdash_c \bigcirc_c (\top \downarrow e \Rightarrow \text{request}_c(n, s(n), e)) \]  \hspace{1cm} (20)

which is the same assertion as the rule \( \text{IR} \) but for the self subtrace. Then, by the classical temporal logic, the three non-temporal implications in the assumptions of the rule \( \text{INVLSe} \) can be generalized to strong implications. For example, the first assumption of the rule \( \text{INVLSe} \) can be generalized to

\[ \vdash_c \bigcirc_c (\top \downarrow e \land \text{request}_c(n, s(n), e) = (s'(n), \text{ors}, \text{ois})) \Rightarrow A \]  \hspace{1cm} (21)

From Eq. 20 and Eq. 21, we immediately have

\[ \vdash_c \bigcirc_c (\top \downarrow e \Rightarrow A) \]  \hspace{1cm} (22)

Therefore, the request event \( \top \downarrow e \) in Eq. 17 can be rewritten to \( A \). The other disjuncts can be similarly rewritten. The result is \( \bigcirc_c A \) that is the conclusion of the rule \( \text{INVLSe} \). Similarly, the rule \( \text{INVL} \) that we saw in the overview section (in Fig. 6) is in fact derived by applying the rule \( \text{SInv} \) to the rule \( \text{INVLSe} \) (in Fig. 12).

(Rule \( \text{Init} \)): The rule \( \text{Init} \) states that at the beginning of the execution, the state \( s \) of every node \( n \) is the state defined by the init function of the implementation. (Rule \( \text{PostPre} \)): The rule \( \text{PostPre} \) states that in the self subtrace, the post-state \( s' \) of every event is the pre-state \( s \) of the next event.

We note that this assertion does not hold on the whole trace as the events of different components are interleaved. (Rule \( \text{Seq} \)): The rule \( \text{Seq} \) (for state equality) states that if the stepping node \( n \) is not a node \( n \), then the state of \( n \) stays unchanged ie. its pre-state \( s(n) \) and post-state \( s'(n) \) are equal.

The above four rules derive inductive inference rules for the state of the top component. Let us consider the derived rule \( \text{INVSSE}' \) in Fig. 12. (Rule \( \text{INVSSE}' \)): The rule \( \text{INVSSE}' \) states that if a state predicate \( S \) holds in the initial state and all the three handler functions, request, indication and periodic preserve \( S \), then \( S \) always holds in the self events. This rule is used to prove state invariants. Similar to the rule \( \text{INVLSe} \), this rule reduces a global temporal assertion to local non-temporal assertions about the handler functions. To derive this rule, the rule \( \text{Init} \) is used in the base case. For the inductive case, the rule \( \text{PostPre} \) brings the invariant \( S \) on the post-state of an event to the pre-state of the next event. To show that each step preserves the invariant, there are two cases. If the node is not stepping, the rule \( \text{Seq} \) is used. If it is, expanding self in the rule \( \text{ASelf} \) leads to a case-analysis for the handler functions that directly proved by the assumptions of the rule \( \text{INVSSE}' \).

(Rule \( \text{RSeq} \)): The rule \( \text{RSeq} \) (for round sequence) states that the round numbers are non-decreasing.

Messages are transmitted using basic links at the leaves of the stack. A basic link is a weak communication primitive; it can drop, reorder and duplicate messages. The next three rules, \( \text{GST} \), \( \text{FDup} \) and \( \text{NForge} \), state the properties of basic links. Stronger communication primitives can be programmed based on these properties. (Rule \( \text{GST} \)): The rule \( \text{GST} \) states that after the round \( r_{GST} \)
In the next section, we will consider the semantics of partially synchronous networks and GST. (Rule NForge): The rule NForge states that links do not forge messages. If a message is sent to a correct node, it is delivered in the same round. (Rule UniOR): The rule UniOR (for unique output request) states that if a request is issued at most once, then it is delivered at most once. If an output request \( e \) is issued at most once at the current event and it is not issued at any other event, then if \( e \) is executed at an event, it is never issued before or after that event. The rule UniOI (unique output indication) states a similar fact for indications and is elided.

(Rule Node): The rule Node states that the current node \( n \) is always in the set of executing nodes \( N \). This rule is used to reason about the subsets of nodes such as quorums. (Rule Quorum): The derived rule Quorum states that assuming that the number of correct nodes is more than \( t_1 \), if the size of a subset \( N \) of nodes (called a quorum) is more than \( t_2 \) and the sum of \( t_1 \) and \( t_2 \) is more than the total number of nodes, then there is at least one correct node in \( N \). Usually \( t_1 \) and \( t_2 \) are both half the number of nodes \( N \). Intuitively, this rule holds because the two sets are large enough to have at least one common element. Quorums are the basis of many distributed protocols and the Quorum rule presents an intuitive reasoning principle for them. We illustrate the use of the quorum rules in verification of the uniform reliable broadcast in the appendix [Appendix 2020] § 2.

Global Stabilization Time (GST) if a message is sent to a correct node, it is delivered in the same round. It axiomatizes the partial synchrony of the network and is used to prove liveness properties. (We will consider the semantics of partially synchronous networks and GST in the next section.) In particular, the rule GST is used to show that the eventual failure detector component eventually suspects no correct node. The rule GST also derives the rule Floss (in Fig. 12). (Rule Floss): The rule Floss (for fair-loss) states that links are fair in dropping messages in the sense that they do not systematically drop any particular message. If a node sends a message infinitely often and the receiver is correct, then the message is delivered to the receiver infinitely often. The rule Floss is used to verify stubborn links that are implemented on top of basic links. (Rule FDup): The rule FDup (for finite duplication) states that basic links duplicate a message only a finite number of times. If the same message is delivered to a node infinitely often, then it is sent infinitely often. (Rule NForge): The rule NForge (for no-forge) states that links do not forge messages. If a message is delivered, it is previously sent.

Fig. 12. A subset of the TLC Derived Inference Rules. \( S \) is a predicate on \( \text{State}_c \). \( N \) is a set.
6 DISTRIBUTED STACK SEMANTICS

In this section, we present the semantics of distributed components. It is a novel operational semantics that models the interaction of composed components in partially synchronous networks. The transitions are labeled with traces of events. Thus, the operational semantics leads to a trace semantics for composed components. TLC is proved sound with respect to this semantics in § 7.

Given a stack \( S \) of components, the semantics defines the transitions that nodes deploying \( S \) take. It models propagation and processing of downward request and periodic events, and upward indication events across layers of components. It also models the crash-stop failure of nodes, and propagation, loss and duplication of messages. After a node crashes, it does not take any steps; a node is called correct if it does not crash.

The semantics models partially synchronous networks. In synchronous networks, a fixed upper bound on message delivery time is known. In contrast, no such bound exists for asynchronous networks and many distributed computing abstractions including consensus are impossible in this model [Fischer et al. 1985]. However, most practical networks including the Internet fall in the partially synchronous model. In these networks, either a bound holds but is not known a priori, or a bound eventually holds. Partial synchrony [Dwork et al. 1988] presented the basic round model for partially synchronous networks. Our semantics follows the basic round model. In this model, each round consists of sending, delivering and processing messages. In each round, only a subnet of sent messages may be delivered; the rest are lost. However, after a round \( r_{GST} \) called the Global Stabilization Time (GST), every message that is sent to a correct node is delivered in the same round. After this round, the network stabilizes and protocols can rely on its synchrony. In practice, the network will eventually remain stable long enough for the protocol to achieve its goal.

The variables used in the operational semantics are defined in Fig. 13. We denote the set of node identifiers \( n \) by \( \mathbb{N} \). As mentioned before, we uniquely identify each component in a stack by the (reverse) list of branch numbers in the path from the top to that component. With the reverse list, moving up and down the tree corresponds to adding and removing an index at the head of the list. We call this sequence the distinct location \( d \in D \) of the component. The substack at location \( d \) of a stack \( S \) is denoted by \( S(d) \). The definition of each component declares the component state type \( \text{State}_c \). The state of a distributed component \( s \in S \) is a mapping from \( \mathbb{N} \) to \( \text{State}_c \). The state of a distributed stack \( \sigma \in \Sigma \) is a heterogeneous map from \( D \) to \( S \) types. A message is a tuple of the sender node, the receiver node, the location of the receiver component and the message payload \( m \). We use \( ms \in \mathcal{M} \) to denote a multi-set of messages. The state of the transition system \( w \in \mathcal{W} \) (for world) is a tuple \( (\sigma, ms, f, r) \) where \( \sigma \) is the state of the distributed stack, \( ms \) is the multi-set (or bag) of in-transit messages, \( f \) is the set of failed nodes and \( r \) is the round number. Given a stack \( S \), the initial state \( w_0(S) \) maps the state of every node.

![Fig. 13. Operational Semantics Variables](image-url)
and location to the state that the init function of the component at that location returns, sets the initial bag of messages \( ms \) and failed nodes \( f \) to the empty set, and sets the round to the initial round \( r_0 \). We use \( e \) to denote user-level events such as send\(_i\)\((n,m)\). The orientation \( o \) of an event is either \( \downarrow \) for request, \( \uparrow \) for indication or \( \perp \) for periodic events.

An event label \( l \) is a record \((n,r,d,o,fe,\sigma,\sigma',or,oi)\) where \( n \) is the stepping node, \( r \) is the round and \( d \) is the location where the event is executed, \( o \) is the orientation of the event, \( fe \) is either fail or an executed user event \( e \), \( \sigma \) and \( \sigma' \) are the stack pre-state and post-state, \( or \) is the issued request event and \( oi \) is the issued indication event. (To access the fields of a label, we use functions with the same names as fields.) An output request event \( or \) is a tuple \((i,e)\) of the target subcomponent number \( i \) and the user event \( e \). (We note that to present a core semantics, the request, indication and periodic handler functions of the components return one rather than a list of request and indication events. A list of events can be similarly processed in sequence. We also elide the complication that \( o, or \) and \( oi \) are option values.) A trace \( \tau \) is a sequence of label events. The \( i \)-th event of a trace \( \tau \) is denoted by \( \tau(i) \). The trace \( \tau_{i..j} \) denotes the sub-trace of \( \tau \) from and including location \( i \) to and excluding location \( j \), and the trace \( \tau_{i..} \) denotes the sub-trace of \( \tau \) from and including location \( i \) onward. Given a predicate \( p \) on event labels, the sub-trace \( \tau_p \) is the projection of \( \tau \) for events that satisfy \( p \). We use the overline notation to denote multiple instances; for example, we use \( \overline{\tau}_i \) to denote multiple traces \( \tau_i \) one for each index \( i \) in the context. The trace \( \tau \cdot \tau' \) denotes the concatenation of the traces \( \tau \) and \( \tau' \), and \( \overline{\tau} \) denotes the concatenation of the traces \( \tau_i \).

Given a stack of components \( S \), Fig. 14 presents the operational semantics for \( S \). The semantics is parametric in the round \( r_{GST} \) (Global Stabilization Time). We start with an overview. A round \( \rightarrow \) comprises two parts. (1) The first part \( \rightarrow^{req}_t \) (with \( t \) for top-level) is a finite sequence of (a) top-level request transitions (and their following request and indication transitions) and (b) node failure transitions. Send request transitions at the leaf layers result in messages. (2) The second part \( \rightarrow^{req}_p \) (with \( p \) for periodic) is a transition that delivers and processes messages and executes the periodic handlers. The two parts result is a round transition \( \rightarrow \rightarrow^{req}_t \rightarrow^{req}_p \). The trace semantics \( T(S) \) of \( S \) is the set of traces \( \tau \) of infinite transitions \( \overline{\rightarrow^{req}_p} \) starting from the initial state \( w_0(S) \). We consider infinite traces to reason about liveness properties.

In the rules, we use \( _- \) as a placeholder for variables that are not used in the context. The two rules \( REQUEST \) and \( FAIL \) make top-level transitions \( \rightarrow_t \). The rule \( REQUEST \) uses the helper transition \( \rightarrow^{req} \). The transition \( \rightarrow^{req} \) processes request events; it is taken by the two rules \( REQ \) and \( REQ' \). The rule \( REQ \) uses the helper transition \( \rightarrow^{ind} \). The transition \( \rightarrow^{ind} \) processes indication events; it is taken by the two rules \( IND \) and \( IND' \). The rule \( IND \), in turn, uses the helper transition \( \rightarrow^{req} \). The transitions \( \rightarrow^{req} \) and \( \rightarrow^{ind} \) are interdependent; the rule \( REQ \) that makes the transition \( \rightarrow^{req} \) uses the transition \( \rightarrow^{ind} \), and the rule \( IND \) that makes the transition \( \rightarrow^{ind} \) uses the transition \( \rightarrow^{req} \).

The rule \( PERIODIC \) makes periodic transitions \( \rightarrow_p \). It uses the helper transitions \( \rightarrow^{msg} \) and \( \rightarrow^{per} \). The transition \( \rightarrow^{msg} \) that processes messages is taken by the rule \( MSG \). The transition \( \rightarrow^{per} \) that executes the periodic functions is taken by the rules \( PER \) and \( PER' \). Next, we take a closer look at each rule.

The rule \( FAIL \) makes a top-level transition with a fail event for a node that has not already failed, and adds it to the set of failed nodes \( f \). Similarly, the other rules require that the executing node has not already failed.

The rule \( REQUEST \) executes a top-level request. It makes a transition \( \rightarrow_t \), if a request transition \( \rightarrow^{req} \) can be taken with a trace starting with a top-level (request) event. The top-level request, in turn, may result in a sequence of requests and indications. The rule \( REQ \) makes transitions \( \rightarrow^{req} \) for request events on the internal (non-leaf) components. We take a close look at the rule \( REQ \).
and the other rules are similar to it. The first event of the trace represents that at a node \( n \) and a component at location \( d \), a request \( \downarrow \) event \( e \) is executed that takes the pre-state of the stack \( \sigma \) to the post-state \( \sigma' \), issues the request event \( e_1 \) to the \( i \)-th subcomponent and issues the indication event \( e_2 \). Let \( c \) be the top component of the stack at location \( d \). The state \( s \) for the component \( c \) is obtained from the stack state \( \sigma \). The request function \( \text{request}_c \) is called with the node identifier \( n \), the pre-state for the node \( s(n) \) and the request event \( e \), and results in the post-state \( s'_{n} \) for \( n \), the request event \( (i,e_1) \), i.e., the request \( e_1 \) for the \( i \)-th subcomponent, and the indication event \( e_2 \). The state \( s \) of the component is updated to \( s' \) with the new state \( s'_{n} \) for the node \( n \), and the state of the stack \( \sigma \) is updated to \( \sigma' \) with the new state \( s' \) at location \( d \). The issued request event \( (i,e_1) \) inductively results in a transition \( \rightarrow_{\text{req}} \) at the \( i \)-th substack whose location is \( i : d \). Similarly the issued indication event \( e_2 \) inductively results in a transition \( \rightarrow_{\text{ind}} \) at location \( d \) on the parent component. The trace for the whole transition is the original request event concatenated with the two subsequent transition traces for the issued request and indication events.

Basic links are used as the leaves of a stack. the rule \( \text{Req}' \) makes a transition for a send request on a leaf component. The rule adds tuples to the in-transit messages that contain the sender and the receiver node identifiers, the component location and the message payload. A message can be duplicated in the network. Therefore, the overline notation indicates a finite number of duplicate message. Further, we note that messages are added to an unordered multi-set of messages. Therefore, messages can be arbitrarily reordered.

The rule \( \text{Ind} \) makes a transition \( \rightarrow_{\text{ind}} \) for indication events from components except the topmost. (We note that the location of the first event in the trace label of this rule is \( i : d \) to exclude the top.) The rule is similar to the rule \( \text{Req} \) in structure but executes an indication instead of a request event. The indication event is executed at the parent component that is at location \( d \). The rule \( \text{Ind}' \) makes a transition \( \rightarrow_{\text{ind}} \) for indication events from the topmost component. In this case, there is no explicit parent component; thus, the rule simply records the issued indication in its label.

The rule \( \text{Periodic} \) delivers messages and executes periodic functions. It drops messages sent to failed nodes \( f \) (using the drop function). In addition, before the round \( r_{\text{GST}} \), it may drop some other messages and retain a subset of messages \( ms' \). The set of remained messages are delivered by the message transition \( \rightarrow_{\text{msg}} \), and calls to the periodic handlers are started at the top level by the period transition \( \rightarrow_{\text{per}} \). The rule \( \text{Msg} \) that makes the transition \( \rightarrow_{\text{msg}} \) delivers all the messages in its pre-state. For every message, it issues a delivery indication event at the recipient node and component location. The trace of the transition is the concatenation of the traces of the indication transitions for all the messages. The rule \( \text{Per} \) executes the periodic function of a component at a (non-leaf) location \( d \) and recursively for every subcomponent \( i \) at location \( i : d \). The rule \( \text{Per}' \) is similar to the rule \( \text{Req} \) in structure but in addition to calling the periodic function on the component, it propagates the periodic calls to lower-level components as well. At the leaf layers, the rule \( \text{Per}' \) makes trivial periodic transitions.

7 SOUNDNESS OF TLC

In this section, we define the semantics of assertions on execution traces and prove the soundness of TLC with respect to distributed stack semantics (that we saw in § 6). We note that TLC is independent of the distributed stack semantics and its soundness can be studied for other semantics.

We define a model \( m \) as a tuple \((\tau, i, l)\) of a trace \( \tau \), a position \( i \) in the trace and an interpretation \( l \). A trace \( \tau \) is the sequence of event labels of an execution. To evaluate temporal operators, the model includes a position \( i \) in the trace. The model also includes an interpretation \( l \) that maps free rigid variables, and interpreted functions and predicates to concrete values, functions and predicates. We define the set of models \( M(S) \) of a stack \( S \) as the set of tuples \((\tau, 0, l_0)\) where \( \tau \) is a trace of \( S \), \( 0 \) is the first position and \( l_0 \) is an initial interpretation that includes mappings for
Fig. 14. Semantics of Distributed Stacks.
evaluates the term \( t \) to the value \( v \). We remember from the classical temporal logic [Manna and Pnueli 1992] that a rigid variable has the same value in all elements of a trace, while a flexible variable may assume distinct values in different elements. The rule \( \text{VARM} \) evaluates a rigid variable \( x \) using the interpretation \( I \). On the other hand, separate rules evaluate the flexible variables \( n, r, d, o, e, s, s', \) \( \text{ois} \) and \( \text{ors} \) based on \( r(i) \), the event at the \( i \)-th position in the trace \( \tau \). For instance, the rule \( \text{NM} \) evaluates the flexible variable \( n \) for the current node to the first element \( n \) of \( r(i) \). Let us take a closer look at the rule \( \text{SM} \) that evaluates the flexible variable \( s \) (pre-state). We remember from the distributed stack semantics that if an event at location \( d \) is a request or periodic event, then it is applied to the component at location \( d \), but if it is an indication event, it is applied to the parent component at location \( \text{tail}(d) \). Therefore, if the event at location \( d \) has a downward orientation, i.e., \( \downarrow \) or \( \not\downarrow \), then the location \( d \) is applied to the stack state \( \sigma \) to obtain the component state. However, if it has an upward orientation, i.e., \( \uparrow \) then the location \( \text{tail}(d) \) is applied to \( \sigma \). The rule \( \text{SM}' \) for the post-state \( s' \) is similar. The rule \( \text{CM} \) evaluates constants except \( \text{Correct} \) by the interpretation \( I \). The rule \( \text{CSM} \) evaluates the constant \( \text{Correct} \) to the set of nodes in \( N \) that do not have a fail transition in the trace. As the pre-state \( s \) and post-state \( s' \) variables take function values, not only a function but also a variable can be applied to terms. The two applications are evaluated in \( \text{FunM} \) and \( \text{FunM}' \) respectively.

The rule \( \text{PREDM} \) evaluates predicates using the interpretation \( I \). The definition of the models relation for conjunction, negation and quantification is standard. The strict always operator \( \not\text{\bigcirc} \) requires the assertion to hold in every future position starting from the position after the current. The strict always in the past operator \( \not\text{\bigcirc}_d \) requires a similar condition in the past. The strict eventual operator \( \not\text{\bigtriangledown} \) requires the assertion to hold in at least one future position starting from the position after the current. The strict eventual in the past operator \( \not\text{\bigtriangledown}_d \) requires a similar condition in the past. As we defined the non-strict temporal operators as syntactic sugar for strict ones, their semantics are derived from the above semantics. The next operator \( \bigcirc \) requires the assertion to hold in the position after the current. As defined at the bottom of Fig. 15, we say that an event label \( \ell \) is on the self component \( \text{mself}(i) \), if it is a request (\( \downarrow \)) or periodic (\( \not\downarrow \)) at the top location (\( \{i\} \)) or is an indication (\( \uparrow \)) at the second level (at location \( \{i\} \) for some \( i \). The self operator \( \otimes \) requires the assertion to hold on the self subtrace, i.e., the projection of the trace over \( \text{mself} \). More precisely, the self operator requires the assertion to hold on the first self position after the current position.

We are now ready to state the soundness of TLC. An assertion \( A \) is valid for a stack \( S \), written as \( \models_S A \), if and only if every model of \( S \) satisfies \( A \).

**Definition 4 (Valid Assertion).** For all \( S \) and \( A \), \( \models_S A \) iff for all \( m \in M(S) \), \( m \models A \).

We say that a set of assertions \( \Gamma \) entail an assertion \( A \) if and only if every model \( m \) of \( S \) that models \( \Gamma \) also models \( A \).

**Definition 5 (Models Relation).** For all \( \Gamma \), \( S \) and \( A \), \( \models_S A \) iff for all \( m \in M(S) \), if \( m \models \Gamma \) then \( m \models A \).

The following theorem states the soundness of TLC. If assuming assertions \( \Gamma \), TLC derives an assertion \( A \) then \( \Gamma \) entails \( A \).

**Theorem 2 (Soundness).** For all \( \Gamma \), \( S \), \( c, \overline{S}' \), \( A \), such that \( S = \text{stack}(c, \overline{S}') \), if \( \Gamma \vdash_c A \), then \( \Gamma \models_S A \).

The detailed proofs are available in the appendix [Appendix 2020] § 5.1. The following corollary is immediately derived. It states that if assuming valid assertions, TLC derives an assertion then that assertion is valid as well. In other words, TLC derives only valid assertions from valid assertions.

**Corollary 2.** For all \( \Gamma \), \( c, S, \overline{S}' \), \( A \) such that \( S = \text{stack}(c, \overline{S}') \), if \( \models_S \Gamma \) and \( \Gamma \vdash_c A \), then \( \models_S A \).
The ultimate goal of this project is mechanized distributed middleware. This goal is a huge un-
proof terms. These approaches proved unsuccessful due to the syntactic nature of proofs in
programming language to capture component definitions and its Ltac proof language to construct
TLC’s compositionality and applicability. We have finished all the proofs of the components in the
appendix to ensure that
is comprehensive. Nonetheless, we have been mechanizing the proofs
in Coq. The earliest
in Coq. The earliest
in Coq. The earliest

8 MECHANIZATION

The ultimate goal of this project is mechanized distributed middleware. This goal is a huge un-
taking and fully achieving it may take multiple years. The main topic of this paper is TLC,
its compositionality and applicability. We have finished all the proofs of the components in the
appendix to ensure that TLC is comprehensive. Nonetheless, we have been mechanizing the proofs
in Coq. The TLC Coq framework provides a deep embedding of an enriched lambda calculus for
defining functional components, an evaluation engine for embedded terms, an inductive definition
of TLC, and a set of tactics for constructing TLC proof terms. We have used this library to success-
fully mechanize the verification of the stubborn link and the perfect link components and we are
extending mechanization to the other components.

Embedding Approaches. We tried different approaches for encoding TLC in Coq. The earliest
attempts were shallow embeddings of TLC. The intent was to utilize Coq’s Gallina functional
programming language to capture component definitions and its Ltac proof language to construct
proof terms. These approaches proved unsuccessful due to the syntactic nature of proofs in TLC.
The syntactic rules of TLC require recursive analysis of the syntax of terms, which cannot be done directly on Gallina terms. We define a deep embedding of a minimal functional programming language to program component terms. This embedding is an untyped lambda calculus enriched with pattern matching terms, externally defined functions, value literals, value constructors, and locally nameless parameters. Similarly, we define a deep embedding of the syntax of TLC as well.

**Embedding.** The syntax of terms is defined as the inductive type presented in Fig. 16.(a). The TParameter, TAbstraction, and TApplication terms come directly from untyped lambda calculus. We adopted the locally nameless representation [Charguéraud 2012] to separately define parameters and variables. We chose this encoding instead of implementing capture-avoiding substitution of arbitrarily named variables. Coq requires all recursive functions to be structurally decreasing. Algorithms for capture-avoiding substitution are not strictly decreasing and are rejected by Coq. Bound variables, represented by the TParameter constructor, are referenced using deBruijn indices. Free variables, represented by the TVariable constructor, are named strings.

The TConstructor term represents a constructor of an inductive type. The TLiteral term represents a literal value of a Coq type, such as the natural numbers. The TFunction term represents a function that is not defined explicitly in the term language, such as recursively defined functions. These terms are lifted into Coq, evaluated, and the result is lowered into the embedded term type. The TMatch constructor represents a pattern matching expression. The TFailure term is the empty term, produced when an ill-formed term is evaluated.

To simplify the definition of terms, we have defined a library of Coq notations for the term language. The library allows for the relatively direct translation of the implementation of the components into embedded terms. Similarly, Coq notations are provided for the assertion language and the sequent judgements. These notations allow writing judgements in the commonly used form. The context of a sequent is the set of variable names that are universally quantified along with the list of assumed assertions. As an example, Fig. 16.(b) shows the statement of the stubborn delivery property: if a message is sent, it is infinitely often delivered. The context declares the list of free variable n, n’ and m and no assumed assertions [::]. The conclusion is read as follows: if two nodes n and n’ are correct and n at the top level [] sends a request event -> to send the message m to n’, then infinitely often at n’, at the top level [], the indication event <- that delivers the message m from n is executed.

**Logic.** The basic rules and axioms of TLC are encoded as an inductive type. Fig. 16.(c) shows three constructors that are representative of the encoding of the rules and axioms of TLC. The rules are extended with rules specific to the implementation of the extended term language. The first constructor, DAEvaluateP, states that if the terms within the first premise can be simplified then proving the conclusion assuming the premise can be reduced to proving the conclusion assuming the simplified premise. The [[A Ap]] notation represents assertion evaluation, which replaces all computational terms within an assertion with the terms produced by their evaluation. Terms are evaluated recursively inside of a monad, which produces a failure case when a failure term is evaluated. The second constructor, DSCut, is a sequent logic rule that can be used to introduce an assertion as a premise. The third constructor, DPIR, is an axiom of the TLC program logic, the rule IR that we saw in § 5.

The framework provides tactics to facilitate applying TLC. For example, it provides a set of tactics that mirror a subset of Coq’s Ltac tactics for producing Gallina proof terms, as well as a library of lemmas. These tactics allow proofs to be written in a more natural, Coq-like style. These are tactics such as d_left (d refers to the derives relation), d_right, d_splitp (p for premise), and d_splitc (c for conclusion), which mirror the primitive Coq tactics left, right, destruct (on conjunctive hypotheses), and split. In addition to these basic tactics, there are tactics that automate some multi-step common tasks. For example, the d_evalc imitates the Coq simpl tactic.
evaluating all terms in the conclusion assertion, and the \(d\) have tactic automates the application of the DSCut rule that we saw above.

9 RELATED WORK


Temporal logic [Manna and Pnueli 1992] is a modal logic that can abstract and reason about time. It can be used to state and check properties of programs specially reactive programs [Alur et al. 2004; Cave et al. 2014; Cook et al. 2011; Das et al. 2018; Jeffrey 2012]. TLA (Temporal Logic of Actions) [Lamport 1994, 2000] is a logic for description, specification and verification of distributed protocols. The transition system of a protocol can be described as action assertions and its specification can be written as temporal logic assertions [Manna and Pnueli 1992]. It has been used [Chaudhuri et al. 2010; Lamport 2002] for model checking [Newcombe et al. 2015] and interactive verification [Chand et al. 2016] of distributed systems. A TLA protocol is described as a monolithic transition system. In contrast, TLC defines event interfaces between components and supports their composition. More importantly, it supports compositional verification of components. In addition, in contrast to TLA that requires the protocol to be described as a transition system, TLC supports...
functional implementation of protocols that can be directly executed. Further, in contrast to TLA, TLC defines an operational semantics to support the soundness of the logic.

I/O Automata [Lynch and Tuttle 1989] models specifications and protocols as transition systems and provides simulation proof techniques [Lynch and W. Vaandrager 1995] between automata. In contrast, TLC captures component implementations as functional programs and specifications as descriptive temporal assertions, and provides a program logic and a compositional proof technique to derive the specification for the implementation.

Both I/O Automata [Lynch and Tuttle 1989] and Reactive Modules [Alur and Henzinger 1999] model protocols as transition systems. They capture specifications in the semantic domain as either transitions systems or properties of execution traces. In contrast, TLC captures component implementations as functional programs and specifications as temporal assertions. I/O Automata provides simulation proof techniques [Lynch and W. Vaandrager 1995] between automata. The simulation proofs are written in the semantic domain. In contrast, TLC provides a program logic to derive the specification for the implementation. Both I/O Automata and and Reactive Modules support composition of interacting modules with matching input and outputs. They support assume-guarantee reasoning where each module can be verified based on the specification of the other module. TLC models distributed systems as structured stacks of components where each component composes with its subcomponents below and its parent component above. Similar to the assume-guarantee reasoning, it supports compositional verification of each component based on the specification of its subcomponents. However, no assumptions for the parent component is needed. The specification of each component is for the most general parent. A verified component can serve as the subcomponent of any parent component.

EventML [Rahli 2012] is a functional domain-specific language for distributed protocols. Protocols written in EventML can be translated to Nuprl [Constable et al. 1986] and then interactively verified. It has been used to verify monolithic replicated services [Rahli et al. 2018; Schiper et al. 2014]; however, it does not address compositional verification.

IronFleet [Hawblitzel et al. 2015] models a distributed system as a hierarchy of state transition systems at multiple levels of abstraction from the high-level specification to the low-level implementation. It proves a refinement [He et al. 1986; Lynch and W. Vaandrager 1995] between a layer and the layer immediately above it. However, it only considers monolithic protocols without subcomponents and the verification is carried out using refinement in contrast to a program logic.

Similarly, network refinement [Koh et al. 2019] presents specifications for a swap server that can be both tested and verified using observational refinement. The server is well-integrated with several other verified systems. To contrast, TLC is a temporal logic and can verify liveness in addition to safety properties. Further, TLC focuses on composition of distributed protocols and builds component stacks on basic links that are much weaker than TCP.

Verdi [Wilcox et al. 2015; Woos et al. 2016] models several network semantics and provides transformations from correct protocols in one semantics to another. It has been applied to verification of state machine replication. Similar to TLC, Verdi provides a form of vertical composition. However, its proofs are based on simulation [He et al. 1986; Lynch and W. Vaandrager 1995] in the semantic domain rather than a program logic.

Chapar [Lesani et al. 2016] presents an operational semantics and a proof technique for verification of causally consistent distributed stores. Similar to TLC, Chapar considers the interface between clients and store implementations; however, only for causal consistency. Further, verification is based on simulation rather than program logic.

PSync [Dragoi et al. 2016] is a DSL for distributed protocols based on the heard-of round-based model [Charron-Bost and Schiper 2009]. This lockstep model enables proof automation that has
been successfully applied to verification of consensus variants. However, PSync left composition as future work.

Ivy [Padon et al. 2016] is an interactive tool that assists in finding inductive invariants. It has been applied to verification of a few distributed protocols. A follow-up work [Taube et al. 2018] lets the user split a protocol into logical modules with explicit invariants. Modules facilitate an assume-guarantee reasoning such that verification of each falls in a separate decidable theory. While the main focus of Ivy is automatic verification of separate parts of monolithic protocols, TLC’s focus is compositional verification of stacks of protocols. Further, in contrast to Ivy, TLC supports verification of liveness properties.

Disel [Sergey et al. 2017; Wilcox et al. 2017] is a program logic for distributed protocols that provides Floyd-Hoare-style specification [Floyd 1967; Hoare 1969; Reynolds 2002] and proof rules for horizontal composition. In Disel, the specification of a component is written in terms of its state and the message pool. On the other hand, in TLC, the temporal specification is written in terms of the interface events almost verbatim from the natural language description. Disel and Hoare-style reasoning require definition of stable invariants and sometimes ghost variables. However, TLC does not require additional annotations on the components. Disel can state and prove safety properties while TLC can state and prove both safety and liveness properties. A follow-up work [García-Pérez et al. 2018] similarly applies the rely-guarantee reasoning to verification of a decomposition of Paxos [Boichat et al. 2003]. However, it does not consider the leader election subcomponent. This paper considers leader election and epoch change as well.

Also related is recent work by Merten et al. [Merten et al. 2018] on Cage, a system for verifying the complexity and optimality of distributed systems in domains like routing and load balancing. Rather than verifying such systems using a program logic, as in TLC, Cage builds protocols and associated implementations that are correct by construction by recasting distributed systems as games with close-to-optimal equilibria, which it then executed using a distributed implementation of no-regret dynamics.

10 CONCLUSION

TLC is a temporal program logic for compositional verification of stacks of distributed components. Its assertion language can capture both safety and liveness properties. Using a transformation function that lowers the specification of components to be used as subcomponents, TLC supports compositional verification of components based on only the specification of their subcomponents. It features intuitive inference rules and induction principles that can deduce assertions about a component based on its functional implementation. TLC and the lowering transformation are proved sound with respect to the operational semantics of distributed stacks in partially synchronous networks. TLC has been successfully applied to verify a stack of fundamental distributed components as the first steps towards certified distributed system stacks.

We hope that the design of TLC motivates further exploration for compositional verification methods in the rich space beyond the traditional structure of the Floyd-Hoare logic. In particular, concurrent and distributed program logics that directly support reasoning about the execution order of events seem to capture intuitive proofs.

ACKNOWLEDGMENTS

We thank Gordon Stewart for reading the draft and providing insightful comments. We thank Peng (Perry) Wang and Jason Gross for crucial help with the subtleties of Coq.
REFERENCES


REFERENCES


