TLC: Temporal Logic of Distributed Components

JEREMIAH GRIFFIN, University of California, Riverside, USA
MOHSEN LESANI, University of California, Riverside, USA
NARGES SHADAB, University of California, Riverside, USA
XIZHE YIN, University of California, Riverside, USA

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.

123

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:
Jeremiah Griffin, Mohsen Lesani, Narges Shadab, and Xizhe Yin. 2020. TLC: Temporal Logic of Distributed Components. *Proc. ACM Program. Lang.* 4, ICFP, Article 123 (August 2020), 30 pages. https://doi.org/10.1145/3409005

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].

Authors' addresses: Jeremiah Griffin, University of California, Riverside, USA, fhous001@ucr.edu; Mohsen Lesani, University of California, Riverside, USA, lesani@cs.ucr.edu; Narges Shadab, University of California, Riverside, USA, nshad001@ucr.edu; Xizhe Yin, University of California, Riverside, USA, xyin014@ucr.edu.

This work is licensed under a Creative Commons Attribution International 4.0 License.
Link to use: https://creativecommons.org/licenses/by/4.0/

2475-1421/2020/8-ART123
https://doi.org/10.1145/3409005

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

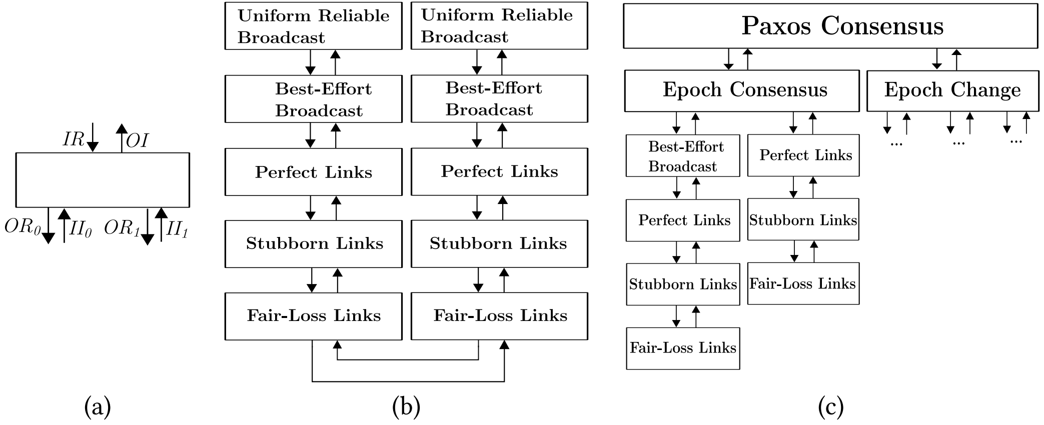

Fig. 1. (a) Events. IR: Input Request, OR: Output request, II: Input Indication, OI: Output Indication (b) Uniform Reliable Broadcast stack at two nodes and (c) Paxos Consensus stack

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. 2. A component is parametric for the type of the events at the top and the bottom of the component as depicted in Fig. 1.(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 $\overline{OR}$ to and the input indications of types $\overline{II}$ 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. 2 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 $\mathsf{send}_{\mathsf{l}}(n, m)$ of type $\mathsf{Req}_{\mathsf{l}}$ to send message m to node n and issue indications $\mathsf{deliver}_{\mathsf{l}}(n, m)$ of type $\mathsf{Ind}_{\mathsf{l}}$ 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.

$$
\begin{array}{l}
\mathsf{Comp}\ \mathit{IR}\ \ \mathit{OI}\ \ \overline{(\mathit{OR}, \mathit{II})} :=\\
\quad \langle\, \mathsf{State}\colon \mathsf{Type},\\
\quad\ \ \mathsf{init}\colon \mathbb{N} \to \mathsf{State},\\
\quad\ \ \mathsf{let\ Out} = \mathsf{State} \times \mathsf{List}\ (\Sigma\ \overline{\mathit{OR}}) \times \mathsf{List}\ \mathit{OI}\ \mathsf{in}\\
\quad\ \ \mathsf{request}\colon \mathbb{N} \times \mathsf{State} \times \mathit{IR} \to \mathsf{Out},\\
\quad\ \ \mathsf{indication}\colon \mathbb{N} \times \mathsf{State} \times \Sigma\ \overline{\mathit{II}} \to \mathsf{Out},\\
\quad\ \ \mathsf{periodic}\colon \mathbb{N} \times \mathsf{State} \to \mathsf{Out}\, \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Stack}\colon \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type} :=\\
\mid \mathsf{stack}\colon \mathsf{Comp}\ \mathit{IR}\ \mathit{OI}\ \ \overline{(\mathit{OR}, \mathit{II})}\ \times\\
\qquad\quad \prod\ (\overline{\mathsf{Stack}\ \mathit{OR}\ \mathit{II}}) \to\\
\qquad\quad \mathsf{Stack}\ \mathit{IR}\ \mathit{OI}\\
\mid \mathsf{link}\colon \mathsf{Stack}\ \mathsf{Req}_{\mathsf{l}}\ \mathsf{Ind}_{\mathsf{l}}
\end{array}
$$

Fig. 2. Component and Stack. *IR*: input requests type, *OI*: output indications type, $\overline{\mathit{OR}}$: output requests types (one per subcomponent), $\overline{\mathit{II}}$: 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.

As Fig. 1.(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. 1.(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. 1.(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

```
PLC: Component Req_pl Ind_pl (Req_sl, Ind_sl) :=
L1   let slc := 0 in
L2   ⟨State := ⟨counter: Nat,
L3              received: Set[⟨ℕ, Nat⟩]⟩,
L4    init := λn. ⟨0, ∅⟩,
L5
L6    request := λ n, s, ir.
L7        let ⟨c, r⟩ := s in
L8        match ir with
L9        | send_pl(n′, m) ⇒
L10           let c′ := c + 1 in
L11           let or := (slc, send_sl(n′, ⟨c′, m⟩)) in
L12           ⟨⟨c′, r⟩, [or], []⟩,
L13       end
L14
L15   indication := λ n′, s, ii.
L16       let ⟨c, r⟩ =: s in
L17       match ii with
L18       | (slc, deliver_sl(n, ⟨c′, m⟩) ⇒
L19           if (⟨n, c′⟩ ∈ r)
L20               ⟨s, [], []⟩
L21           else
L22               let r′ := r ∪ {⟨n, c′⟩} in
L23               let oi := deliver_pl(n, m) in
L24               ⟨⟨c, r′⟩, [], [oi]⟩
L25       end
L26
L27   periodic := λ n, s. ⟨s, [], []⟩ ⟩
```
(a)

SL_1 (Stubborn delivery):
$n \in \mathsf{Correct} \wedge n' \in \mathsf{Correct} \rightarrow$
$(n \bullet \top \downarrow \mathsf{send}_{\mathsf{sl}}(n', m)) \Rightarrow \Box\Diamond(n' \bullet \top \uparrow \mathsf{deliver}_{\mathsf{sl}}(n, m))$
If a correct node n sends a message m to a correct node n', then n' delivers m infinitely often.

SL_2 (No-forge):
$(n \bullet \top \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m)) \leftsquigarrow (n' \bullet \top \downarrow \mathsf{send}_{\mathsf{sl}}(n, m))$
If a node n delivers a message m with sender n', then n' previously sent m to n.

(b)

PL_1 (Reliable delivery):
$n \in \mathsf{Correct} \wedge n' \in \mathsf{Correct} \rightarrow$
$(n \bullet \top \downarrow \mathsf{send}_{\mathsf{pl}}(n', m) \rightsquigarrow (n' \bullet \top \uparrow \mathsf{deliver}_{\mathsf{pl}}(n, m))$
If a correct node n sends a message m to a correct node n', then n' will eventually deliver m.

PL_2 (No-duplication):
$[n' \bullet \top \downarrow \mathsf{send}_{\mathsf{pl}}(n, m) \Rightarrow \hat{\boxminus}\neg(n' \bullet \top \downarrow \mathsf{send}_{\mathsf{pl}}(n, m))] \rightarrow$
$[n \bullet \top \uparrow \mathsf{deliver}_{\mathsf{pl}}(n', m) \Rightarrow \hat{\boxminus}\neg(n \bullet \top \uparrow \mathsf{deliver}_{\mathsf{pl}}(n', m))]$
If a message is sent at most once, it will be delivered at most once.

PL_3 (No-forge):
$(n \bullet \top \uparrow \mathsf{deliver}_{\mathsf{pl}}(n', m)) \leftsquigarrow (n' \bullet \top \downarrow \mathsf{send}_{\mathsf{pl}}(n, m))$
If a node n delivers a message m with sender n', then n' previously sent m to n.

(c)

The assertion $(n \bullet \top \downarrow \mathsf{send}_{\mathsf{sl}}(n', m))$ describes an event at node n at the top level interface $\top$ for the request $\downarrow$ to send the message m to the node n', i.e., $\mathsf{send}_{\mathsf{sl}}(n', m)$. The assertion $(n \bullet \top \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m))$ describes an event at node n at the top level interface $\top$ for the indication $\uparrow$ to deliver the message m from the node n', i.e., $\mathsf{deliver}_{\mathsf{sl}}(n', m)$. Correct is the set of nodes that have not crashed.

Fig. 3. (a) Perfect Link Component PLC. (b) Stubborn Links Specification. (c) Perfect Links Specification.

message is sent together with the new counter value using the stubborn link subcomponent (at L_{11}-L_{12}). Upon a delivery indication of a message from the stubborn link subcomponent (at L_{15}-L_{18}), if the message is already received, it is ignored (at L_{19}-L_{20}). Otherwise, the message identifier is added to the received set and a delivery indication event is issued (at L_{21}-L_{24}). (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 L_{27}).

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 $\downarrow$, $\wr$ and $\uparrow$ 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 $[]$, for its left child C_2 is $[0]$ and for its right grandchild C_5 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 $[]$ from the client. When e_1 is processed in the top component C_1 at $[]$, 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 $[]$, finally, e_6 that is an indication $\uparrow$ at the top level $[]$, 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 $\Box\mathcal{A}$ states that the assertion $\mathcal{A}$ holds at every event in the future including the current event. The always in the past assertion $\boxminus\mathcal{A}$ states that $\mathcal{A}$ holds at every event in the past including the current event. The eventually assertion $\Diamond\mathcal{A}$ states that $\mathcal{A}$ holds at some future event. The eventually in the past assertion $\diamondminus\mathcal{A}$ states that $\mathcal{A}$ holds at some past event. The strict versions $\hat{\Box}$, $\hat{\boxminus}$, $\hat{\Diamond}$ and $\hat{\diamondminus}$ exclude the current event. The strong implication $\mathcal{A} \Rightarrow \mathcal{A}'$ is syntactic sugar for $\Box(\mathcal{A} \rightarrow \mathcal{A}')$ where $\rightarrow$ is the logical implication. The leads-to assertion $\mathcal{A} \rightsquigarrow \mathcal{A}'$ is syntactic sugar for $\Box(\mathcal{A} \rightarrow \Diamond\mathcal{A}')$. Similarly, the preceded-by assertion $\mathcal{A} \leftsquigarrow \mathcal{A}'$ is syntactic sugar for $\Box(\mathcal{A} \rightarrow \diamondminus\mathcal{A}')$. An assertion is non-temporal if it does not include any temporal operators.

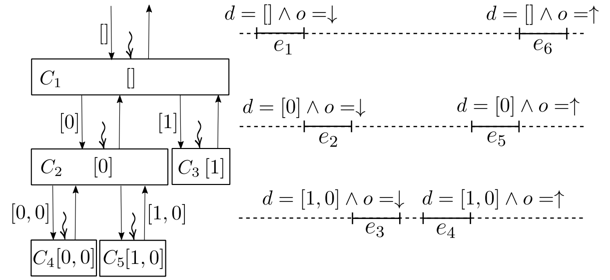

Fig. 4. Semantics of Component Stacks.

The user events are the event objects that the protocol handlers take as argument and issue, for example $\mathsf{send}_{\mathsf{pl}}(n, m)$. 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 **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 $n \bullet \mathcal{A}$ (where $\bullet$ is used as a separator) is sugar for $\mathbf{n} = n \wedge \mathcal{A}$; it describes an event that is executed at node n and satisfies $\mathcal{A}$. The syntactic sugar assertion $\top\, o\, e$ stands for $\mathbf{d} = [] \;\wedge\; \mathbf{o} = o \;\wedge\; \mathbf{e} = e$; it describes an event that is at the top ($\top$) level interface $[]$, its orientation is o (either the constant $\downarrow$ for requests, $\wr$ for periodics or $\uparrow$ for indications) and its user event is e. For example, the assertion $(n \bullet \top \downarrow \mathsf{send}_{\mathsf{sl}}(n', m))$ describes an event at node n at the top level interface $[]$ where the request ($\downarrow$) event $\mathsf{send}_{\mathsf{sl}}(n', m)$ 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 $\mathbf{d} = [i] \;\wedge\; \mathbf{o} = o \;\wedge\; \mathbf{e} = e$; 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 $(n \bullet 1 \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m))$ describes an event at node n at the interface location $[1]$ where the indication ($\uparrow$) event $\mathsf{deliver}_{\mathsf{sl}}(n', m)$ 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 SL_1 states that once a message is sent, it is delivered infinitely often. The no-forge property SL_2 states that a stubborn link never forges a message. The properties SL_1 and SL_2 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 PL_1 states that perfect links can reliably transmit messages between correct nodes. The no-duplication property PL_2 states that perfect links do not redundantly deliver messages. (It is notable that $(p \Rightarrow \hat{\boxminus}\neg p) \rightarrow (p \Rightarrow \hat{\Box}\neg p)$; hence the $\hat{\Box}$ conjunct is omitted in the no-duplication property.) The no-forge property PL_3 states that perfect links do not forge messages. The property PL_1 is a liveness and the properties PL_2 and PL_3 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 $\top$ 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 $\downarrow$ for request and $\uparrow$ 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 SL_2 in Fig. 3 can be summarized as the following assertion: $(n \bullet \mathsf{deliver}_{\mathsf{sl}}(n', m)) \leftsquigarrow (n' \bullet \mathsf{send}_{\mathsf{sl}}(n, m))$. 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

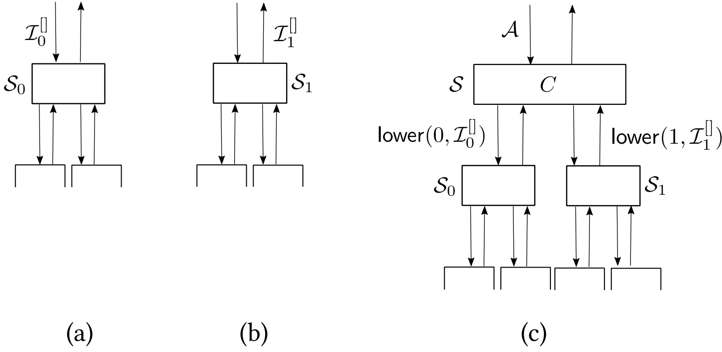

Fig. 5. Illustration of Lowering. (a) Stack $\mathcal{S}_0$ and its specification $\mathcal{I}_0^{[]}$. (b) Stack $\mathcal{S}_1$ and its specification $\mathcal{I}_1^{[]}$. (c) Stack $\mathcal{S}$ that composes $\mathcal{S}_0$ and $\mathcal{S}_1$ as subcomponents. To prove the assertion $\mathcal{A}$ for $\mathcal{S}$, it is sound to assume the lowering of $\mathcal{I}_0^{[]}$ and $\mathcal{I}_1^{[]}$, and derive $\mathcal{A}$ in TLC, i.e., $\overline{\mathsf{lower}(i, \mathcal{I}_i^{[]})} \vdash_c \mathcal{A}$.

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 $\mathcal{S}_i$ is given as an invariant $\mathcal{I}_i^{[]}$ (Fig. 5.(a) and (b)). Consider that we have a stack $\mathcal{S}$ with the component c at the top and the substacks $\overline{\mathcal{S}_i}$, i.e., $\mathcal{S} = \mathsf{stack}(c, \overline{\mathcal{S}_i})$. We want to verify that $\mathcal{S}$ satisfies its specification $\mathcal{A}$ (Fig. 5.(c)). What can we assume for each subcomponents $\mathcal{S}_i$? We define the translation function lower on invariants and show that to prove the validity of $\mathcal{A}$ for $\mathcal{S}$, it is sound to assume $\overline{\mathsf{lower}(i, \mathcal{I}_i^{[]})}$ and derive $\mathcal{A}$ in TLC, i.e., $\overline{\mathsf{lower}(i, \mathcal{I}_i^{[]})} \vdash_c \mathcal{A}$.

The SL_2 assertion is in the invariant sub-language. Applying the lower transformation to SL_2 to use it as the 0-th subcomponent results in the following:

$$\begin{aligned} \mathsf{SL}_2' \;=\; \mathsf{lower}(0, \mathsf{SL}_2) \;=\; & \mathsf{lower}(0, (n \bullet \top \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m)) \rightsquigarrow (n' \bullet \top \downarrow \mathsf{send}_{\mathsf{sl}}(n, m))) \;= \\ & (n \bullet 0 \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m)) \rightsquigarrow (n' \bullet 0 \downarrow \mathsf{send}_{\mathsf{sl}}(n, m)) \end{aligned} \tag{1}$$

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_1 to result in SL_1'.

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

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 Γ, the no-forge property is valid for PLC.

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

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

OR′

$$\vdash_c n \bullet i \downarrow e \;\Rightarrow\; \hat{\diamondminus}(n \bullet (i,e) \in \mathbf{ors} \wedge \mathsf{self})$$

OI′

$$\vdash_c n \bullet \top \uparrow e \;\Rightarrow\; \hat{\diamondminus}(n \bullet e \in \mathbf{ois} \wedge \mathsf{self})$$

InvL

$$\frac{\begin{array}{c} \forall e.\;\; \top \downarrow e \wedge \mathsf{request}_c(\mathbf{n},\mathbf{s}(\mathbf{n}),e) = (\mathbf{s}'(\mathbf{n}),\mathbf{ois},\mathbf{ors}) \rightarrow \mathcal{A} \\ \forall e,i.\;\; i \uparrow e \wedge \mathsf{indication}_c(\mathbf{n},\mathbf{s}(\mathbf{n}),(i,e)) = (\mathbf{s}'(\mathbf{n}),\mathbf{ois},\mathbf{ors}) \rightarrow \mathcal{A} \\ \top \wr \mathsf{per} \wedge \mathsf{periodic}_c(\mathbf{n},\mathbf{s}(\mathbf{n})) = (\mathbf{s}'(\mathbf{n}),\mathbf{ois},\mathbf{ors}) \rightarrow \mathcal{A} \\ \mathcal{A} \text{ non-temporal} \end{array}}{\vdash_c \mathsf{self} \Rightarrow \mathcal{A}}$$

Trans$\diamondminus$

$$(\mathcal{A} \Rightarrow \diamondminus\mathcal{A}' \;\wedge\; \mathcal{A}' \Rightarrow \mathcal{A}'') \rightarrow \mathcal{A} \Rightarrow \diamondminus\mathcal{A}''$$

Trans$\diamondminus\diamondminus$

$$(\mathcal{A} \Rightarrow \diamondminus\mathcal{A}' \;\wedge\; \mathcal{A}' \Rightarrow \diamondminus\mathcal{A}'') \rightarrow \mathcal{A} \Rightarrow \diamondminus\mathcal{A}''$$

Fig. 6. Three Selected TLC Inference Rules and Two Basic Temporal Logic Lemmas

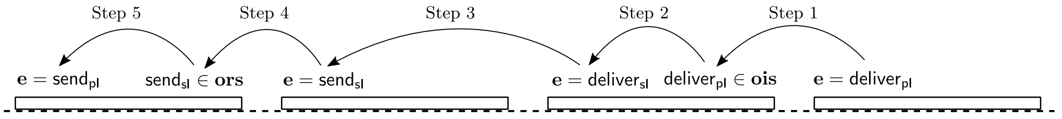

Fig. 7. Illustration of the Proof Steps. The trace is a sequence of events from left to right. Each rectangle represents an event. In a sequence of steps, the proof shows that a perfect link deliver event $\mathsf{deliver}_{\mathsf{pl}}$ is preceded by a perfect link send event $\mathsf{send}_{\mathsf{pl}}$.

(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 issuing event is a perfect link send event. By the transitivity of precedence, it is concluded form the above steps that A perfect link delivery is preceded by a perfect link send.

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$\diamondminus$ and Trans$\diamondminus\diamondminus$. 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 $\downarrow$ 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 $\uparrow$ 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$\diamondminus$ and Trans$\diamondminus\diamondminus$ state basic temporal transitivity properties. By rule OI′, we have

$$\text{Step 1:}\quad \Gamma \vdash_{\mathsf{PLC}} (n \bullet \top \uparrow \mathsf{deliver}_{\mathsf{pl}}(n',m)) \Rightarrow \diamondminus(n \bullet \mathsf{deliver}_{\mathsf{pl}}(n',m) \in \mathbf{ois} \wedge \mathsf{self}) \tag{2}$$

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

$$\mathcal{A} = n \bullet \mathsf{deliver}_{\mathsf{pl}}(n',m) \in \mathbf{ois} \rightarrow \exists c.\; (n \bullet 0 \uparrow \mathsf{deliver}_{\mathsf{sl}}(n',\langle c,m\rangle)) \tag{3}$$

Considering the implementation in Fig. 3.(a), the two cases for request and periodic are straightforward as **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:

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

By Lemma Trans⟐ on Eq. 2 and Eq. 4, and existential elimination for c, we have

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

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

From Γ, and Eq. 1 (lowered SL_2'), instantiating m with $\langle c, m \rangle$ and unfolding $\rightsquigarrow$, we have

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

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

$$\text{Step 4:}\quad \Gamma \vdash_{\text{PLC}} (n' \bullet 0 \downarrow \text{send}_{\text{sl}}(n, \langle c, m \rangle)) \Rightarrow ⟐(n' \bullet (0, \text{send}_{\text{sl}}(n, \langle c, m \rangle)) \in \mathbf{ors} \wedge \text{self}) \tag{7}$$

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

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

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

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

By Lemma Trans⟐ on Eq. 7 and Eq. 8, we have

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

From Lemma Trans⟐⟐ on Eq. 5, Eq. 6 and Eq. 9, we have

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

that is

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

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 $\mathbf{s}$ and post-state $\mathbf{s}'$ 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.

$$
\begin{array}{rcll}
x & := & & \text{Variable} \\
 & | & n \mid d \mid o \mid e & \text{Rigid} \\
 & | & ors \mid ois \mid s \mid i & \\
 & | & \mathbf{n} \mid \mathbf{r} \mid \mathbf{d} \mid \mathbf{o} \mid \mathbf{e} & \text{Flexible} \\
 & | & \mathbf{ors} \mid \mathbf{ois} \mid \mathbf{s} \mid \mathbf{s}' & \\
c & := & & \text{Constant} \\
 & | & [\,] \mid \downarrow \mid \uparrow \mid \wr \mid \text{per} & \\
 & | & \mathbb{N} \mid \text{Correct} & \\
f & := & + \mid :: \mid .. & \text{Function} \\
t & := & & \text{Term} \\
 & | & x \mid c & \\
 & | & f\ (t_1,..,t_n) \mid x\ (t_1,..,t_n) & \text{Function App} \\
p & := & & \text{Predicate} \\
 & | & < \mid = \mid \in \mid \subseteq \mid .. & \\
a & := & p\ (t_1,..,t_n) & \text{Atom} \\
\mathcal{A} & := & a & \text{Assertion} \\
 & | & \mathcal{A} \wedge \mathcal{A} \mid \neg \mathcal{A} & \text{Proposition} \\
 & | & \forall x.\ \mathcal{A} & \text{Quantification} \\
 & | & \hat{\square} \mathcal{A} \mid \hat{\boxminus} \mathcal{A} & \text{Temporal} \\
 & | & \hat{\Diamond} \mathcal{A} \mid \hat{\diamondminus} \mathcal{A} \mid \bigcirc \mathcal{A} & \text{Temporal} \\
 & | & \circledS\, \mathcal{A} & \text{Self Assertion}
\end{array}
$$

Assertion for the stack at location d:

$\mathcal{A}^d := \mathcal{A}$ such that

$a := (\mathbf{n} = t \ \wedge \ \mathbf{d} = d' \ \wedge \ \mathbf{o} = t \ \wedge \ \mathbf{e} = t) \quad d' \supseteq d$

$\quad \mid t \in \text{Correct}$

and $\bigcirc$ and $\circledS$ are not used.

Invariant for the stack at location d:

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

Invariant:

$\mathcal{I} := \square \mathcal{A}$ such that

$\bigcirc$ and $\circledS$ are not used.

Syntactic Sugar:

$$
\begin{array}{rcl}
n \bullet \mathcal{A} & \triangleq & \mathbf{n} = n \ \wedge \ \mathcal{A} \\
\top\ o\ e & \triangleq & \mathbf{d} = [\,] \ \wedge \ \mathbf{o} = o \ \wedge \ \mathbf{e} = e \\
i\ o\ e & \triangleq & \mathbf{d} = [i] \ \wedge \ \mathbf{o} = o \ \wedge \ \mathbf{e} = e \\
\text{self} & \triangleq & (\mathbf{d} = [\,] \wedge \mathbf{o} = \downarrow) \vee \\
 & & (\mathbf{d} = [\,] \wedge \mathbf{o} = \wr) \vee \\
 & & (\exists i.\ \mathbf{d} = [i] \wedge \mathbf{o} = \uparrow) \\
\square \mathcal{A} & \triangleq & \mathcal{A} \wedge \hat{\square} \mathcal{A} \\
\boxminus \mathcal{A} & \triangleq & \mathcal{A} \wedge \hat{\boxminus} \mathcal{A} \\
\Diamond \mathcal{A} & \triangleq & \mathcal{A} \vee \hat{\Diamond} \mathcal{A} \\
\diamondminus \mathcal{A} & \triangleq & \mathcal{A} \vee \hat{\diamondminus} \mathcal{A} \\
\mathcal{A} \Rightarrow \mathcal{A}' & \triangleq & \square(\mathcal{A} \rightarrow \mathcal{A}') \\
\mathcal{A} \leadsto \mathcal{A}' & \triangleq & \square(\mathcal{A} \rightarrow \Diamond \mathcal{A}') \\
\mathcal{A} \leftsquigarrow \mathcal{A}' & \triangleq & \square(\mathcal{A} \rightarrow \diamondminus \mathcal{A}')
\end{array}
$$

Fig. 8. Assertion Language

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 $\bigcirc$ and $\circledS$. The next operator $\bigcirc$ 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 $\circledS$ allows stating assertions about the self subtrace. The assertion $\circledS\, \mathcal{A}$ asserts $\mathcal{A}$ 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 $\square \mathcal{A}^d$. To support lowering, the atomic assertions used in an assertion $\mathcal{A}^d$ are constrained to have the form $\mathbf{n} = t \ \wedge \ \mathbf{d} = d' \ \wedge \ \mathbf{o} = t' \ \wedge \ \mathbf{e} = t''$ such that $d' \supseteq d$. The location variable $\mathbf{d}$ 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 $\mathbf{n}$, the orientation $\mathbf{o}$ and the user-level event $\mathbf{e}$. Further, invariant assertions do not use the next $\bigcirc$ or self $\circledS$ operators.

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. A fundamental question is how the specification of a component

Definition 1 (Lowering Assertions). $lower(i, \mathcal{I}^{[]})$

$$lower(i, \mathcal{I}^{[]}) \triangleq restrict(\boldsymbol{d} \supseteq [i],\ push(i, \mathcal{I}^{[]}))$$

Definition 2 (Pushing an Assertion). $push(i, \mathcal{A}^{[]})$:

$$
\begin{array}{rcl}
push(i, \boldsymbol{n} = t_1 \wedge \boldsymbol{d} = d \wedge \boldsymbol{o} = t_2 \wedge \boldsymbol{e} = t_3) & \triangleq & \boldsymbol{n} = t_1 \wedge \boldsymbol{d} = i ::: d \wedge \boldsymbol{o} = t_2 \wedge \boldsymbol{e} = t_3 \\
push(i, t \in Correct) & \triangleq & t \in Correct \\
push(i, \mathcal{A}_1^{[]} \wedge \mathcal{A}_2^{[]}) & \triangleq & push(i, \mathcal{A}_1^{[]}) \wedge push(i, \mathcal{A}_2^{[]}) \\
push(i, \neg \mathcal{A}^{[]}) & \triangleq & \neg push(i, \mathcal{A}^{[]}) \\
push(i, \forall x.\ \mathcal{A}^{[]}) & \triangleq & \forall x.\ push(i, \mathcal{A}^{[]}) \\
push(i, \hat{\boxdot} \mathcal{A}^{[]}) & \triangleq & \hat{\boxdot} push(i, \mathcal{A}^{[]}) \\
push(i, \hat{\boxminus} \mathcal{A}^{[]}) & \triangleq & \hat{\boxminus} push(i, \mathcal{A}^{[]}) \\
push(i, \hat{\Diamond} \mathcal{A}^{[]}) & \triangleq & \hat{\Diamond} push(i, \mathcal{A}^{[]}) \\
push(i, \hat{\diamondminus} \mathcal{A}^{[]}) & \triangleq & \hat{\diamondminus} push(i, \mathcal{A}^{[]})
\end{array}
$$

Definition 3 (Restricting an Assertion). $restrict(\mathcal{A}', \mathcal{A})$:

$$
\begin{array}{rcl}
restrict(\mathcal{A}', a) & \triangleq & a \\
restrict(\mathcal{A}', \mathcal{A}_1 \wedge \mathcal{A}_2) & \triangleq & restrict(\mathcal{A}', \mathcal{A}_1) \wedge restrict(\mathcal{A}', \mathcal{A}_2) \\
restrict(\mathcal{A}', \neg \mathcal{A}) & \triangleq & \neg restrict(\mathcal{A}', \mathcal{A}) \\
restrict(\mathcal{A}', \forall x.\ \mathcal{A}) & \triangleq & \forall x.\ restrict(\mathcal{A}', \mathcal{A}) \\
restrict(\mathcal{A}', \hat{\boxdot} \mathcal{A}) & \triangleq & \hat{\boxdot}(\mathcal{A}' \rightarrow restrict(\mathcal{A}', \mathcal{A})) \\
restrict(\mathcal{A}', \hat{\boxminus} \mathcal{A}) & \triangleq & \hat{\boxminus}(\mathcal{A}' \rightarrow restrict(\mathcal{A}', \mathcal{A})) \\
restrict(\mathcal{A}', \hat{\Diamond} \mathcal{A}) & \triangleq & \hat{\Diamond} restrict(\mathcal{A}', \mathcal{A}) \\
restrict(\mathcal{A}', \hat{\diamondminus} \mathcal{A}) & \triangleq & \hat{\diamondminus} restrict(\mathcal{A}', \mathcal{A})
\end{array}
$$

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

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 $\mathcal{S}_i$ is given as an invariant $\mathcal{I}_i^{[]}$ (Fig. 5.(a) and (b)). We have a stack $\mathcal{S}$ with the component c at the top and the substacks $\overline{\mathcal{S}_i}$, i.e., $\mathcal{S} = \text{stack}(c, \overline{\mathcal{S}_i})$. We want to verify that $\mathcal{S}$ satisfies its specification $\mathcal{A}$ (Fig. 5.(c)). We define the translation function lower on invariants and show that to prove the validity of $\mathcal{A}$ for $\mathcal{S}$, it is sufficient to assume $\overline{\text{lower}(i, \mathcal{I}_i^{[]})}$ and derive $\mathcal{A}$ in TLC. Fig. 9 represents the the function lower on the invariant sub-language $\mathcal{I}^{[]}$. 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 $\mathcal{I}^d := \Box \mathcal{A}^d$, the location values $d' \supseteq d$ are explicit in assertions $\mathcal{A}^d$. Given a top-level assertion $\mathcal{A}^{[]}$ 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.

$$\mathsf{SL}_2' \;=\; \mathsf{lower}(0, \mathsf{SL}_2) \;= \tag{10}$$

$$\mathsf{lower}(0, (n \bullet \top \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m)) \leftsquigarrow (n' \bullet \top \downarrow \mathsf{send}_{\mathsf{sl}}(n, m))) \;= \tag{11}$$

$$\mathsf{restrict}(\mathbf{d} \supseteq [0],\ \mathsf{push}(0,\ \Box[(n \bullet \top \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m)) \rightarrow \diamondminus(n' \bullet \top \downarrow \mathsf{send}_{\mathsf{sl}}(n, m))])) \;= \tag{12}$$

$$\mathsf{restrict}(\mathbf{d} \supseteq [0],\ \Box[(n \bullet 0 \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m)) \rightarrow \diamondminus(n' \bullet 0 \downarrow \mathsf{send}_{\mathsf{sl}}(n, m))]) \;= \tag{13}$$

$$\Box[\mathbf{d} \supseteq [0] \rightarrow (n \bullet 0 \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m)) \rightarrow \diamondminus(n' \bullet 0 \downarrow \mathsf{send}_{\mathsf{sl}}(n, m))] \;= \tag{14}$$

$$\Box[(n \bullet 0 \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m)) \rightarrow \diamondminus(n' \bullet 0 \downarrow \mathsf{send}_{\mathsf{sl}}(n, m))]] \;= \tag{15}$$

$$(n \bullet 0 \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m)) \leftsquigarrow (n' \bullet 0 \downarrow \mathsf{send}_{\mathsf{sl}}(n, m)) \tag{16}$$

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 $\Box\mathcal{A}$ for a stack. After pushing the assertion to the i-th substack, the resulting assertion $\Box\mathsf{push}(i, \mathcal{A})$ does not necessarily remain valid because although the assertion $\mathsf{push}(i, \mathcal{A})$ 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 $\Box\mathsf{push}(i, \mathcal{A})$ is translated to $\Box(\mathbf{d} \supseteq [i] \rightarrow \mathsf{push}(i, \mathcal{A}))$. (The assertion $\mathsf{push}(i, \mathcal{A})$ 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 $\Diamond\mathcal{A}$ 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 SL_2 that we saw in Eq. 1. The property SL_2 is in the invariant language $\mathcal{I}^{[]}$ 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 $\mathbf{d} = []$ to $\mathbf{d} = [0]$. Thus, in the syntactic sugar, $\top$ is translated to 0. In Eq. 14, the syntactic sugar $(n \bullet 0 \uparrow \mathsf{deliver}_{\mathsf{sl}}(n', m))$ includes the conjunct $\mathbf{d} = [0]$. From basic propositional logic, for any $\mathcal{A}$ and $\mathcal{A}'$, we have that $\mathbf{d} \supseteq [0] \rightarrow ((\mathbf{d} = [0] \wedge \mathcal{A}) \rightarrow \mathcal{A}')$ simplifies to $(\mathbf{d} \supseteq [0] \wedge \mathbf{d} = [0] \wedge \mathcal{A}) \rightarrow \mathcal{A}'$. Since $\mathbf{d} = [0]$ is stronger than $\mathbf{d} \supseteq [0]$, it further simplifies to $(\mathbf{d} = [0] \wedge \mathcal{A}) \rightarrow \mathcal{A}'$.

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 $\mathbf{e} = \mathsf{send}(n, m) \Rightarrow m > 0$ for a stack $\mathcal{S}_1$ that states that all the messages that it sends are positive. This assertion is too general for a stack $\mathcal{S}$ that composes $\mathcal{S}_1$ as a subcomponent because the top component or the sibling subcomponents may send negative messages. Further, if the assertions included the next operator $\bigcirc$, 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 $(\mathbf{d} = [] \wedge \mathbf{o} = \downarrow) \Rightarrow \bigcirc(\mathbf{d} = [] \wedge \mathbf{o} = \uparrow)$ for a stack $\mathcal{S}_1$ that states that every top-level request is immediately followed by a indication. The pushed assertion $(\mathbf{d} = [1] \wedge \mathbf{o} = \downarrow) \Rightarrow \bigcirc(\mathbf{d} = [1] \wedge \mathbf{o} = \uparrow)$ is not valid for a stack $\mathcal{S}$ that composes $\mathcal{S}_1$ 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 Ⓢ cannot be lowered. For example, consider the assertion Ⓢ$(\forall n.\ \mathbf{s}(n) > 0)$ for a stack $\mathcal{S}_1$ that states that the state of the top component of $\mathcal{S}_1$ 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 $\mathcal{I}_i^{[]}$ is valid for the stack $\mathcal{S}_i$ and $\mathcal{S}_i$ is a substack of the stack $\mathcal{S}$, then the lowered invariant $\text{lower}(i,\mathcal{I}_i^{[]})$ is valid for $\mathcal{S}$. We use the validity judgement $\vDash_{\mathcal{S}} \mathcal{A}$ that states that $\mathcal{A}$ is valid in every trace of $\mathcal{S}$. (Validity is defined more precisely in § 7.) The detailed proofs are available in the appendix [Appendix 2020] § 5.2.

Theorem 1. *For all $\mathcal{S}$, c, and $\overline{\mathcal{S}_i}$, such that $\mathcal{S} = stack(c,\overline{\mathcal{S}_i})$, if $\vDash_{\mathcal{S}_i} \mathcal{I}_i^{[]}$ then $\vDash_{\mathcal{S}} lower(i,\mathcal{I}_i^{[]})$.*

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 $\Gamma \vdash_c \mathcal{A}$ where Γ is the assumed assertions and $\mathcal{A}$ is the deduced assertion. Consider valid top-level invariants $\overline{\mathcal{I}_i^{[]}}$ for stacks $\overline{\mathcal{S}_i}$, and a stack $\mathcal{S}$ built by the component c on top of $\overline{\mathcal{S}_i}$. The following theorem states that assuming the lowered invariants $\overline{\text{lower}(i,\mathcal{I}_i^{[]})}$, any assertion that TLC deduces for c is valid for $\mathcal{S}$.

Corollary 1 (Composition Soundness). *For all $\mathcal{S}$, c, and $\overline{\mathcal{S}_i}$ such that $\mathcal{S} = stack(c,\overline{\mathcal{S}_i})$, if $\overline{\vDash_{\mathcal{S}_i} \mathcal{I}_i^{[]}}$ and $\overline{lower(i,\mathcal{I}_i^{[]})} \vdash_c \mathcal{A}$ then $\vDash_{\mathcal{S}} \mathcal{A}$.*

5 TLC INFERENCE 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. 1.(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 $\Gamma \vdash_c \mathcal{A}$ where c is the component, Γ is a set of assumed assertions and $\mathcal{A}$ 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_c handler function is called at that step. The request_c function of the component c represents a relation between its inputs: the stepping node $\mathbf{n}$, the pre-state $\mathbf{s}(\mathbf{n})$ of $\mathbf{n}$, and the user event e, and its outputs: the post-state $\mathbf{s}'(\mathbf{n})$, the issued output requests **ors** and the issued output indications **ois**. The rules II (for input indication) and Pe (for periodic) are similar. (Rule II): The rule II states that if an indication $\uparrow$ event e from the i-th subcomponent is executed then the indication_c handler function is called. (An indication event e from a subcomponent i is passed to the indication_c function as the sum term (i, e).) (Rule Pe): The rule Pe states that if at the top $\top$, a periodic $\wr$ event per is executed then the periodic_c 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
$\vdash_c \top \downarrow e \;\Rightarrow\; (\mathbf{s}'(\mathbf{n}), \mathbf{ors}, \mathbf{ois}) = \mathsf{request}_c(\mathbf{n}, \mathbf{s}(\mathbf{n}), e)$

II
$\vdash_c i \uparrow e \;\Rightarrow\; (\mathbf{s}'(\mathbf{n}), \mathbf{ors}, \mathbf{ois}) = \mathsf{indication}_c(\mathbf{n}, \mathbf{s}(\mathbf{n}), (i, e))$

Pe
$\vdash_c \top \wr \mathsf{per} \;\Rightarrow\; (\mathbf{s}'(\mathbf{n}), \mathbf{ors}, \mathbf{ois}) = \mathsf{periodic}_c(\mathbf{n}, \mathbf{s}(\mathbf{n}))$

OR
$\vdash_c n \bullet (i, e) \in \mathbf{ors} \wedge \mathsf{self} \;\Rightarrow\; \hat{\Diamond}(n \bullet i \downarrow e)$

OI
$\vdash_c n \bullet e \in \mathbf{ois} \wedge \mathsf{self} \;\Rightarrow\; \hat{\Diamond}(n \bullet \top \uparrow e)$

OR′
$\vdash_c n \bullet i \downarrow e \;\Rightarrow\; \hat{\diamondminus}(n \bullet (i, e) \in \mathbf{ors} \wedge \mathsf{self})$

OI′
$\vdash_c n \bullet \top \uparrow e \;\Rightarrow\; \hat{\diamondminus}(n \bullet e \in \mathbf{ois} \wedge \mathsf{self})$

APer
$\vdash_c n \in \mathsf{Correct} \;\leftrightarrow\; \Box\Diamond(n \bullet \top \wr \mathsf{per})$

ASelf
$\vdash_c ⓢ\, \Box\, \mathsf{self}$

SInv
$\vdash_c (ⓢ\, \mathcal{I}) \;\leftrightarrow\; \mathsf{restrict}(\mathsf{self}, \mathcal{I})$

Init
$\vdash_c ⓢ\, (\mathbf{s} = \lambda n.\, \mathsf{init}_c(n))$

PostPre
$\vdash_c ⓢ\, (\mathbf{s}' = s \;\Leftrightarrow\; \bigcirc\, \mathbf{s} = s)$

SEq
$\vdash_c \mathbf{n} \neq n \;\Rightarrow\; \mathbf{s}'(n) = \mathbf{s}(n)$

RSeq
$\vdash_c \mathbf{r} = r \;\Rightarrow\; \hat{\boxminus}(\mathbf{r} \leq r)$

GST
$\vdash_c n' \in \mathsf{Correct} \wedge r \geq r_{GST} \wedge$
$(n \bullet d \downarrow \mathsf{send}_{\mathsf{l}}(n', m) \wedge \mathbf{r} = r) \Rightarrow$
$\Diamond(n' \bullet d \uparrow \mathsf{deliver}_{\mathsf{l}}(n, m) \wedge \mathbf{r} = r)$

FDup
$\vdash_c \Box\Diamond(n' \bullet d \uparrow \mathsf{deliver}_{\mathsf{l}}(n, m)) \rightarrow$
$\Box\Diamond(n \bullet d \downarrow \mathsf{send}_{\mathsf{l}}(n', m))$

NForge
$\vdash_c (n' \bullet d \uparrow \mathsf{deliver}_{\mathsf{l}}(n, m)) \Rightarrow$
$\diamondminus(n \bullet d \downarrow \mathsf{send}_{\mathsf{l}}(n', m))$

Node
$\vdash_c \Box\, \mathbf{n} \in \mathbb{N}$

UniOR
$\vdash_c (\mathsf{occ}(\mathbf{ors}, e) \leq 1 \wedge$
$\hat{\boxminus}(\mathbf{n} = n \wedge \mathsf{self} \rightarrow (i, e) \notin \mathbf{ors}) \wedge$
$\hat{\Box}(\mathbf{n} = n \wedge \mathsf{self} \rightarrow (i, e) \notin \mathbf{ors})) \Rightarrow$
$(n \bullet i \downarrow e) \Rightarrow$
$\hat{\boxminus}\neg(n \bullet i \downarrow e) \wedge \hat{\Box}\neg(n \bullet i \downarrow e)$

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

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 $\mathcal{I}$ holds in the self subtrace if and only if the restriction of the assertion $\mathsf{restrict}(\mathsf{self}, \mathcal{I})$ 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 InvLSe (in Fig. 12). (Rule InvLSe): The rule InvLSe states that if a non-temporal assertion holds for all the three handler functions, request, indication and periodic, then the assertion holds in every

self event. We note that this rule reduces a *global temporal assertion* down to local non-temporal *proof obligations about the handler functions*. Let us see how the the rule InvLSe is derived. First, the self assertion in the rule ASelf is expanded to the disjunction of three request, indication or periodic events:

$$\vdash_c \circledS\, \Box(\exists e.\ (\top \downarrow e)\ \vee\ (i \uparrow e)\ \vee\ (\top \wr \text{per})) \tag{17}$$

Then, for the three cases, the rule SInv is applied to the assertions that the rules IR, II and 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 IR can be restated as

$$\vdash_c \Box[\text{self} \rightarrow (\top \downarrow e\ \rightarrow\ (\mathbf{s}'(\mathbf{n}), \mathbf{ors}, \mathbf{ois}) = \text{request}_c(\mathbf{n}, \mathbf{s}(\mathbf{n}), e))] \tag{18}$$

that is equivalent to the following assertion. (We remember that $\mathcal{A} \Rightarrow \mathcal{A}'$ is defined as $\Box(\mathcal{A} \rightarrow \mathcal{A}')$.)

$$\vdash_c \text{restrict}(\text{self},\ \top \downarrow e\ \Rightarrow\ (\mathbf{s}'(\mathbf{n}), \mathbf{ors}, \mathbf{ois}) = \text{request}_c(\mathbf{n}, \mathbf{s}(\mathbf{n}), e)) \tag{19}$$

Then, the result of applying the rule SInv is

$$\vdash_c \circledS\, \top \downarrow e\ \Rightarrow\ (\mathbf{s}'(\mathbf{n}), \mathbf{ors}, \mathbf{ois}) = \text{request}_c(\mathbf{n}, \mathbf{s}(\mathbf{n}), e) \tag{20}$$

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

$$\vdash_c \circledS\, (\top \downarrow e \wedge \text{request}_c(\mathbf{n}, \mathbf{s}(\mathbf{n}), e) = (\mathbf{s}'(\mathbf{n}), \mathbf{ois}, \mathbf{ors})) \Rightarrow \mathcal{A} \tag{21}$$

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

$$\vdash_c \circledS\, (\top \downarrow e\ \Rightarrow \mathcal{A}) \tag{22}$$

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

(Rule Init): The rule Init states that at the beginning of the execution, the state $\mathbf{s}$ of every node n is the state defined by the init_c function of the implementation. (Rule PostPre): The rule PostPre states that in the self subtrace, the post-state $\mathbf{s}'$ of every event is the pre-state $\mathbf{s}$ 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 SEq): The rule SEq (for state equality) states that if the stepping node $\mathbf{n}$ is not a node n, then the state of n stays unchanged ie. its pre-state $\mathbf{s}(n)$ and post-state $\mathbf{s}'(n)$ are equal.

The above four rules derive inductive inference rules for the state of the top component. Let us consider the derived rule InvSSe′ in Fig. 12. (Rule InvSSe′): The rule 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 InvLSe, this rule reduces a global temporal assertion to local non-temporal assertions about the handler functions. To derive this rule, the rule Init is used in the base case. For the inductive case, the rule 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 SEq is used. If it is, expanding self in the rule ASelf leads to a case-analysis for the handler functions that directly proved by the assumptions of the rule InvSSe′.

(Rule RSeq): The rule 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, GST, FDup and NForge, state the properties of basic links. Stronger communication primitives can be programmed based on these properties. (Rule GST): The rule GST states that after the round r_{GST}

$$\frac{\begin{array}{c}\forall e.\ \top \downarrow e \wedge \mathsf{request}_c(\mathbf{n}, \mathbf{s}(\mathbf{n}), e) = (\mathbf{s}'(\mathbf{n}), \mathbf{ois}, \mathbf{ors}) \rightarrow \mathcal{A} \\ \forall e, i.\ i \uparrow e \wedge \mathsf{indication}_c(\mathbf{n}, \mathbf{s}(\mathbf{n}), (i, e)) = (\mathbf{s}'(\mathbf{n}), \mathbf{ois}, \mathbf{ors}) \rightarrow \mathcal{A} \\ \top \wr \mathsf{per} \wedge \mathsf{periodic}_c(\mathbf{n}, \mathbf{s}(\mathbf{n})) = (\mathbf{s}'(\mathbf{n}), \mathbf{ois}, \mathbf{ors}) \rightarrow \mathcal{A} \\ \mathcal{A} \text{ non-temporal}\end{array}}{\vdash_c \circledS\, \Box \mathcal{A}}\ \textsc{InvLSe}$$

$$\frac{\begin{array}{c} S(\mathsf{init}_c(n)) \\ \forall s, e, s'.\ S(s) \wedge \mathsf{request}_c(n, s, e) = (s', _, _) \rightarrow S(s') \\ \forall s, i, e, s'.\ S(s) \wedge \mathsf{indication}_c(n, s, (i, e)) = (s', _, _) \rightarrow S(s') \\ \forall s, s'.\ S(s) \wedge \mathsf{periodic}_c(n, s) = (s', _, _) \rightarrow S(s') \end{array}}{\vdash_c \circledS\, \Box S(\mathbf{s}(n))}\ \textsc{InvSSe}'$$

FLoss
$$\vdash_c n' \in \mathsf{Correct} \rightarrow \Box\Diamond(n \bullet d \downarrow \mathsf{send}_{\mathsf{l}}(n', m)) \rightarrow \Box\Diamond(n' \bullet d \uparrow \mathsf{deliver}_{\mathsf{l}}(n, m))$$

Quorum
$$|\mathsf{Correct}| > t_1 \ \vdash_c \ N \subseteq \mathbb{N} \ \wedge \ |N| > t_2 \ \wedge \ t_1 + t_2 \geq |\mathbb{N}| \Rightarrow \exists n.\ n \in N \wedge n \in \mathsf{Correct}$$

Fig. 12. A subset of the TLC Derived Inference Rules. S is a predicate on State_c. N is a set.

(Global Stabilization Time) 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.

(Rule UniOR): The rule UniOR (for unique output request) states that if a request is issued at most once, then it is executed 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 executed 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 $\mathbf{n}$ is always in the set of executing nodes $\mathbb{N}$. 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 $\mathbb{N}$. 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.

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 $\mathcal{S}$ of components, the semantics defines the transitions that nodes deploying $\mathcal{S}$ 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 subset 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 $d \in \mathcal{D}$ of the component. The substack at location d of a stack $\mathcal{S}$ is denoted by $\mathcal{S}(d)$. The definition of each component declares the component state type State_c. The state of a distributed component $s \in S$ is a mapping from $\mathbb{N}$ to State_c. The state of a distributed stack $\sigma \in \Sigma$ is a heterogeneous map from $\mathcal{D}$ 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 (σ, ms, f, r) where σ 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 $\mathcal{S}$, the initial state $w_0(\mathcal{S})$ maps the state of every node

n :	$\mathbb{N}$	Node ID
d :	$\mathcal{D}$ = List Nat	Distinct Location
cs :	State_c	Component State
s :	S = Map $\mathbb{N}$ State_c	Dist. Comp. State
σ :	Σ = Map $\mathcal{D}$ S	Stack state
m :	M	Message Payload
ms :	$\mathcal{M}$ = MultiSet $(\mathbb{N} \times \mathbb{N} \times \mathcal{D} \times M)$	Messages
f :	$\mathbb{N}$	Failed nodes
r :	$\mathcal{R}$	Round
w :	$\mathcal{W} = \Sigma \times \mathcal{M} \times \mathbb{N} \times \mathcal{R}$	World
$w_0(\mathcal{S})$ =	$\langle(\lambda d, n.\ \mathsf{let\ stack}(c, _) = \mathcal{S}(d)\ \mathsf{in}\ \mathsf{init}_c(n)), \varnothing, \varnothing, r_0\rangle$	Initial World
e, oi :	E	User Event
or :	$I\!E$ = Nat $\times E$	Output request
fe :	$F\!E$	Event or Fail
:=	$e \mid \mathsf{fail}$	
o :	O	Orientation
:=	$\downarrow \mid \uparrow \mid \wr$	
ℓ :	$\mathbb{N} \times \mathcal{R} \times \mathcal{D} \times O \times F\!E \times \Sigma \times \Sigma \times I\!E \times E$	Event Label
τ ::=	ℓ^*	Trace

Fig. 13. Operational Semantics Variables

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 $\mathsf{send_l}(n,m)$. The orientation o of an event is either $\downarrow$ for request, $\uparrow$ for indication or $\wr$ 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, σ and σ' 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 τ is a sequence of label events. The i-th event of a trace τ is denoted by $\tau(i)$. The trace $\tau_{i..j}$ denotes the sub-trace of τ from and including location i to and excluding location j, and the trace $\tau_{i..}$ denotes the sub-trace of τ from and including location i onward. Given a predicate p on event labels, the sub-trace $\tau|_p$ is the projection of τ 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 τ_i one for each index i in the context. The trace $\tau \cdot \tau'$ denotes the concatenation of the traces τ and τ', and $\overline{\cdot \tau_i}$ denotes the concatenation of the traces τ_i.

Given a stack of components $\mathcal{S}$, Fig. 14 presents the operational semantics for $\mathcal{S}$. 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 $\xrightarrow{\tau}_t^*$ (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 $\xrightarrow{\tau'}_p$ (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 $\xrightarrow{\tau \cdot \tau'}$. The trace semantics $T(\mathcal{S})$ of $\mathcal{S}$ is the set of traces τ of infinite transitions $\xrightarrow{\tau}^*$ starting from the initial state $w_0(\mathcal{S})$. 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 σ to the post-state σ', 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 σ. The request function $\mathrm{request}_c$ 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 σ is updated to σ' with the new state s' at location d. The issued request event (i, e_1) inductively results in a transition $\rightarrow_{req}$ at the i-th substack whose location is $i :: d$. Similarly the issued indication event e_2 inductively results in a transition $\rightarrow_{ind}$ 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 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 Ind makes a transition $\rightarrow_{ind}$ 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 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 Ind′ makes a transition $\rightarrow_{ind}$ 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 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_{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 $\rightarrow_{msg}$, and calls to the periodic handlers are started at the top level by the period transition $\rightarrow_{per}$. The rule Msg that makes the transition $\rightarrow_{msg}$ 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 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 Per is similar to the rule 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 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 (τ, i, I) of a trace τ, a position i in the trace and an interpretation I. A trace τ 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 I that maps free rigid variables, and interpreted functions and predicates to concrete values, functions and predicates. We define the set of models $M(\mathcal{S})$ of a stack $\mathcal{S}$ as the set of tuples $(\tau, 0, I_0)$ where τ is a trace of $\mathcal{S}$, 0 is the first position and I_0 is an initial interpretation that includes mappings for

Request

$$\frac{n \in \mathbb{N} \setminus \{f\} \qquad (\sigma, ms, f, r) \xrightarrow{\tau}_{req} (\sigma', ms') \qquad \tau = (n, r, [], _, _, _, _, _, _) \cdot \tau'}{(\sigma, ms, f, r) \xrightarrow{\tau}_{t} (\sigma', ms', f, r)}$$

Fail

$$\frac{n \in \mathbb{N} \setminus \{f\}}{(\sigma, ms, f, r) \xrightarrow{(n, r, [], \bot, \mathsf{fail}, \sigma, \sigma, \bot, \bot)}_{t} (\sigma, ms', f \cup \{n\}, r)}$$

Periodic

$$\frac{\begin{cases} ms' = \mathsf{drop}(ms, f) & \text{if } r \geq r_{GST} \\ ms' \subseteq \mathsf{drop}(ms, f) & \text{else} \end{cases} \qquad (\sigma, ms', f, r) \xrightarrow{\tau}_{msg} (\sigma_0, ms_0) \qquad \overline{\dfrac{(\sigma_n, ms_n, f, r+1) \xrightarrow{\tau_n}_{per} (\sigma_{n+1}, ms_{n+1})}{\tau_n = (n, r+1, [], \wr, \mathsf{per}, _, _, _, _) \cdot \tau'_n}}_{n \in \mathbb{N} \setminus \{f\}} \qquad \overline{\sigma_n = \sigma_{n+1} \quad ms_n = ms_{n+1}}_{n \in f}}{(\sigma, ms, f, r) \xrightarrow{\tau \cdot \overline{\tau_n}_{n \in \mathbb{N} \setminus \{f\}}}_{p} (\sigma_{|\mathbb{N}|}, ms_{|\mathbb{N}|}, f, r+1)}$$

Req

$$\frac{\begin{array}{c} n \in \mathbb{N} \setminus \{f\} \qquad \mathcal{S}(d) = \mathsf{stack}(c, _) \qquad \sigma(d) = s \\ \mathsf{request}_c(n, s(n), e) = (s'_n, (i, e_1), e_2) \\ s' = s[n \mapsto s'_n] \qquad \sigma' = \sigma[d \mapsto s'] \\ (\sigma', ms, f, r) \xrightarrow{\tau_1}_{req} (\sigma_1, ms_1) \qquad \tau_1 = (n, r, i :: d, \downarrow, e_1, _, _, _, _) \cdot \tau'_1 \\ (\sigma_1, ms_1, f, r) \xrightarrow{\tau_2}_{ind} (\sigma_2, ms_2) \qquad \tau_2 = (n, r, d, \uparrow, e_2, _, _, _, _) \cdot \tau'_2 \\ \tau = (n, r, d, \downarrow, e, \sigma, \sigma', (i, e_1), e_2) \cdot \tau_1 \cdot \tau_2 \end{array}}{(\sigma, ms, f, r) \xrightarrow{\tau}_{req} (\sigma_2, ms_2)}$$

Req′

$$\frac{n \in \mathbb{N} \setminus \{f\} \qquad \mathcal{S}(d) = \mathsf{link}}{(\sigma, ms, f, r) \xrightarrow{(n, r, d, \downarrow, \mathsf{send}_{\mathsf{l}}(n', m), \sigma, \sigma, \bot, \bot)}_{req} (\sigma', ms \uplus \{\overline{(n, n', d, m)}\})}$$

Ind

$$\frac{\begin{array}{c} n \in \mathbb{N} \setminus \{f\} \qquad \mathcal{S}(d) = \mathsf{stack}(c, _) \qquad \sigma(d) = s \\ \mathsf{indication}_c(n, s(n), (i, e)) = (s'_n, (i', e_1), e_2) \\ s' = s[n \mapsto s'_n] \qquad \sigma' = \sigma[d \mapsto s'] \\ (\sigma', ms, f, r) \xrightarrow{\tau_1}_{req} (\sigma_1, ms_1) \qquad \tau_1 = (n, r, i' :: d, \downarrow, e_1, _, _, _, _) \cdot \tau'_1 \\ (\sigma_1, ms_1, f, r) \xrightarrow{\tau_2}_{ind} (\sigma_2, ms_2) \qquad \tau_2 = (n, r, d, \uparrow, e_2, _, _, _, _) \cdot \tau'_2 \\ \tau = (n, r, i :: d, \uparrow, e, \sigma, \sigma', (i', e_1), e_2) \cdot \tau_1 \cdot \tau_2 \end{array}}{(\sigma, ms, f, r) \xrightarrow{\tau}_{ind} (\sigma_2, ms_2)}$$

Ind′

$$\frac{n \in \mathbb{N} \setminus \{f\}}{(\sigma, ms, f, r) \xrightarrow{(n, r, [], \uparrow, e, \sigma, \sigma, \bot, \bot)}_{ind} (\sigma, ms)}$$

Msg

$$\frac{ms = \{\overline{(n_i, n'_i, d_i, m_i)}_{i \in I}\} \qquad \sigma_0 = \sigma \qquad ms_0 = \emptyset \qquad \overline{\dfrac{(\sigma_i, ms_i, f, r) \xrightarrow{\tau_i}_{ind} (\sigma_{i+1}, ms_{i+1})}{\tau_i = (n'_i, r, d_i, \uparrow, \mathsf{deliver}_{\mathsf{l}}(n_i, m_i), _, _, _, _) \cdot \tau'_i}}_{i \in I}}{(\sigma, ms, f, r) \xrightarrow{\cdot \overline{\tau_i}_{i \in I}}_{msg} (\sigma_{|I|}, ms_{|I|})}$$

Per′

$$\frac{n \in \mathbb{N} \setminus \{f\} \qquad \mathcal{S}(d) = \mathsf{link}}{(\sigma, ms, f, r) \xrightarrow{(n, r, d, \wr, \bot, \sigma, \sigma, \bot, \bot)}_{per} (\sigma, ms)}$$

Per

$$\frac{\begin{array}{c} n \in \mathbb{N} \setminus \{f\} \qquad \mathcal{S}(d) = \mathsf{stack}(c, \overline{\mathcal{S}'}) \qquad k = |\overline{\mathcal{S}'}| \qquad \sigma(d) = s \\ \mathsf{periodic}_c(n, s(n)) = (s'_n, (i, e_1), e_2) \qquad s' = s[n \mapsto s'_n] \qquad \sigma' = \sigma[d \mapsto s'] \\ (\sigma', ms, f, r) \xrightarrow{\tau_1}_{req} (\sigma'', ms'') \qquad \tau_1 = (n, r, i :: d, \downarrow, e_1, _, _, _, _) \cdot \tau'_1 \\ (\sigma'', ms'', f, r) \xrightarrow{\tau_2}_{ind} (\sigma_0, ms_0) \qquad \tau_2 = (n, r, d, \uparrow, e_2, _, _, _, _) \cdot \tau'_2 \\ \overline{(\sigma_i, ms_i, f, r) \xrightarrow{\tau_i}_{per} (\sigma_{i+1}, ms_{i+1})}_{i \in \{0..k-1\}} \qquad \overline{\tau_i = (n, r, i :: d, \wr, _, _, _, _, _) \cdot \tau'_i}_{i \in \{0..k-1\}} \\ \tau = (n, r, d, \wr, \mathsf{per}, \sigma, \sigma', (i, e_1), e_2) \cdot \tau_1 \cdot \tau_2 \cdot \overline{\tau_i}_{i \in \{0..k-1\}} \end{array}}{(\sigma, ms, f, r) \xrightarrow{\tau}_{per} (\sigma_k, ms_k)}$$

Fig. 14. Semantics of Distributed Stacks.

commonly used integer, list and set functions and predicates. The traces of a stack $\mathcal{S}$ is the set of traces of the executions of $\mathcal{S}$ (for any r_{GST}).

In Fig. 15, we define the models relation, $m \vDash \mathcal{A}$, that is read as the model m models or satisfies the assertion $\mathcal{A}$. We also use the models relation for terms, $m \vDash t \colon v$, that is read as the model m

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 VarM evaluates a rigid variable x using the interpretation I. On the other hand, separate rules evaluate the flexible variables $\mathbf{n}$, $\mathbf{r}$, $\mathbf{d}$, $\mathbf{o}$, $\mathbf{e}$, $\mathbf{s}$, $\mathbf{s}'$, **ois** and **ors** based on $\tau(i)$, the event at the i-th position in the trace τ. For instance, the rule NM evaluates the flexible variable $\mathbf{n}$ for the current node to the first element n of $\tau(i)$. Let us take a closer look at the rule SM that evaluates the flexible variable $\mathbf{s}$ (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 $\mathsf{tail}(d)$. Therefore, if the event at location d has a downward orientation, i.e., $\downarrow$ or $\wr$ then the location d is applied to the stack state σ to obtain the component state. However, if it has an upward orientation, i.e., $\uparrow$ then the location $\mathsf{tail}(d)$ is applied to σ. The rule SM$'$ for the post-state $\mathbf{s}'$ is similar. The rule CM evaluates constants except Correct by the interpretation I. The rule CSM evaluates the constant Correct to the set of nodes in $\mathbb{N}$ that do not have a fail transition in the trace. As the pre-state $\mathbf{s}$ and post-state $\mathbf{s}'$ variables take function values, not only a function but also a variable can be applied to terms. The two applications are evaluated in FunM and FunM$'$ respectively.

The rule PredM evaluates predicates using the interpretation I. The definition of the models relation for conjunction, negation and quantification is standard. The strict always operator $\hat{\Box}$ requires the assertion to hold in every future position starting from the position after the current. The strict always in the past operator $\hat{\boxminus}$ requires a similar condition in the past. The strict eventual operator $\hat{\Diamond}$ 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 $\hat{\diamondminus}$ 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 $\circ$ 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 ℓ is on the self component $\mathsf{mself}(\ell)$, if it is a request ($\downarrow$) or periodic ($\wr$) at the top location ($[]$) or is an indication ($\uparrow$) at the second level (at location $[i]$ for some i). The self operator Ⓢ requires the assertion to hold on the self subtrace, i.e., the projection of the trace over 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 $\mathcal{A}$ is valid for a stack $\mathcal{S}$, written as $\vDash_{\mathcal{S}} \mathcal{A}$, if and only if every model of $\mathcal{S}$ satisfies $\mathcal{A}$.

Definition 4 (Valid Assertion). *For all $\mathcal{S}$ and $\mathcal{A}$, $\vDash_{\mathcal{S}} \mathcal{A}$ iff for all $m \in M(\mathcal{S})$, $m \vDash \mathcal{A}$.*

We say that a set of assertions Γ entail an assertion $\mathcal{A}$ if and only if every model m of $\mathcal{S}$ that models Γ also models $\mathcal{A}$.

Definition 5 (Models Relation). *For all Γ, $\mathcal{S}$ and $\mathcal{A}$, $\Gamma \vDash_{\mathcal{S}} \mathcal{A}$ iff for all $m \in M(\mathcal{S})$, if $m \vDash \Gamma$ then $m \vDash \mathcal{A}$.*

The following theorem states the soundness of TLC. If assuming assertions Γ, TLC derives an assertion $\mathcal{A}$ then Γ entails $\mathcal{A}$.

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

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 Γ, c, $\mathcal{S}$, $\overline{\mathcal{S}'}$, $\mathcal{A}$ such that $\mathcal{S} = stack(c, \overline{\mathcal{S}'})$, if $\vDash_{\mathcal{S}} \Gamma$ and $\Gamma \vdash_c \mathcal{A}$, then $\vDash_{\mathcal{S}} \mathcal{A}$.*

Definition 6 (Model relation).

Rule				Condition
VarM	(τ,i,I)	$\models$	$x : I(x)$	*if* x *rigid*
NM	(τ,i,I)	$\models$	$\boldsymbol{n} : n(\tau(i))$	
RM	(τ,i,I)	$\models$	$\boldsymbol{r} : r(\tau(i))$	
DM	(τ,i,I)	$\models$	$\boldsymbol{d} : d(\tau(i))$	
OM	(τ,i,I)	$\models$	$\boldsymbol{o} : o(\tau(i))$	
EM	(τ,i,I)	$\models$	$\boldsymbol{e} : e(\tau(i))$	
SM	(τ,i,I)	$\models$	$\boldsymbol{s} : \sigma(\tau(i))(d')$ *where* $d' = \begin{cases} d(\tau(i)) & \textit{if } o(\tau(i)) = \downarrow \vee o(\tau(i)) = \lightning \\ \textit{tail}(d(\tau(i))) & \textit{else} \end{cases}$	
SM'	(τ,i,I)	$\models$	$\boldsymbol{s}' : \sigma'(\tau(i))(d')$ *where* $d' = \begin{cases} d(\tau(i)) & \textit{if } o(\tau(i)) = \downarrow \vee o(\tau(i)) = \lightning \\ \textit{tail}(d(\tau(i))) & \textit{else} \end{cases}$	
ORSM	(τ,i,I)	$\models$	$\boldsymbol{ors} : ors(\tau(i))$	
OISM	(τ,i,I)	$\models$	$\boldsymbol{ois} : ois(\tau(i))$	
CM	(τ,i,I)	$\models$	$c : I(c)$	*if* $c \neq \textit{Correct}$
CSM	(τ,i,I)	$\models$	$\textit{Correct} : \{n \mid n \in \mathbb{N} \wedge \nexists\, j \geq 0.\ \tau(j) = (n, [\,], \bot, \textit{fail}, _, _, _, _)\}$	
FunM	m	$\models$	$f(t_1,..,t_n) : f'(v_1,..,v_n)$	*if* $I(f) = f',\ m \models t_1 : v_1,\ ..,\ m \models t_n : v_n$
FunM'	m	$\models$	$x(t_1,..,t_n) : f'(v_1,..,v_n)$	*if* $m \models x : f',\ m \models t_1 : v_1,\ ..,\ m \models t_n : v_n$
PredM	m	$\models$	$p(t_1,..,t_n)$	*iff* $I(p) = p'$, $m \models t_1 : v_1,\ ..\ m \models t_n : v_n$, $p'(v_1,..,v_n) = \textit{true}$
AndM	m	$\models$	$\mathcal{A} \wedge \mathcal{A}'$	*iff* $m \models \mathcal{A}$ *and* $m \models \mathcal{A}'$
NotM	m	$\models$	$\neg\mathcal{A}$	*iff* $m \not\models \mathcal{A}$
ForallM	(τ,i,I)	$\models$	$\forall x.\ \mathcal{A}$	*iff* $(\tau,i,I[x \mapsto v]) \models \mathcal{A}$ *for all* $v \in \textit{dom}(I)$
AlwaysM	(τ,i,I)	$\models$	$\hat{\Box}\mathcal{A}$	*iff* $(\tau,j,I) \models \mathcal{A}$ *for all* j, $j > i$
PAlwaysSM	(τ,i,I)	$\models$	$\hat{\boxminus}\mathcal{A}$	*iff* $(\tau,j,I) \models \mathcal{A}$ *for all* j, $0 \leq j < i$
EventualSM	(τ,i,I)	$\models$	$\hat{\Diamond}\mathcal{A}$	*iff* $(\tau,j,I) \models \mathcal{A}$ *there exists* j, $j > i$
PEventualSM	(τ,i,I)	$\models$	$\hat{\diamondminus}\mathcal{A}$	*iff* $(\tau,j,I) \models \mathcal{A}$ *there exists* j, $0 \leq j < i$
NextM	(τ,i,I)	$\models$	$\bigcirc\,\mathcal{A}$	*iff* $(\tau,i+1,I) \models \mathcal{A}$
SelfM	(τ,i,I)	$\models$	Ⓢ $\mathcal{A}$	*iff* $\tau'_1 = \tau_{0\,..\,i-1}|_{\textit{mself}},\ \tau'_2 = \tau_{i\,..}|_{\textit{mself}},$ $\tau' = \tau'_1 \cdot \tau'_2,\ i' = \lvert\tau'_1\rvert,\ (\tau',i',I) \models \mathcal{A}$

$$\textit{mself}(\ell) \triangleq (d(\ell) = [\,] \wedge o(\ell) = \downarrow)\ \vee\ (d(\ell) = [\,] \wedge o(\ell) = \lightning\,)\ \vee\ (\exists i.\ d(\ell) = [i] \wedge o(\ell) = \uparrow)$$

Fig. 15. Models Relation. $m \models \mathcal{A}$ and $m \models t : v$.

8 MECHANIZATION

The ultimate goal of this project is mechanized distributed middleware. This goal is a huge undertaking 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 successfully 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

High-level DSLs, language extensions and tools [Bakst et al. 2017; Biely et al. 2013; Burckhardt et al. 2012; Cejtin et al. 1995; Kato et al. 1993; Ketsman et al. 2019; Killian et al. 2007; Liu et al. 2012; Miller et al. 2016; Salvaneschi et al. 2019; Samanta et al. 2013; Weisenburger et al. 2018] have been used to raise the level of abstraction, and improve the reliability of distributed systems. Model checking has been extensively applied for bounded verification [Dutertre et al. 2018; Jackson 2006; John et al. 2013; Killian et al. 2007; Konnov et al. 2017; Marić et al. 2017; Musuvathi and Engler 2004; Yabandeh et al. 2009; Yang et al. 2009; Zave 2012] of distributed algorithms. Recently, domain specific logics and verification frameworks have gained momentum to establish the absence of bugs.

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

```
Inductive term :=
| TParameter (p : parameter) (* Nameless bound params *)
| TVariable (v : variable) (* Named free variables *)
| TAbstraction (tb : term) (* Function abstraction *)
| TApplication (tf ta : term) (* Function application *)
| TConstructor (c : constr) (* Value constructors *)
| TLiteral (l : literal) (* Value literals *)
| TFunction (f : function) (* External functions *)
| TFailure (* Computation error *)
| TMatch (ta : term) (cs : cases) (* Pattern matching *)
(* Cases of pattern matching *)
with acase :=
| TCase (p : pattern) (t : term)
with cases :=
| TCNil
| TCCons (c : acase) (cs : cases).
```

(a)

```
Theorem SL_1 :
  Context [:: V "m"; V "n'"; V "n"] [::]
  |- stubborn_link, {A:
  correct "n" /\ correct "n'" ->
  on "n", event []-> CSLSend $ "n'" $ "m" =>>
  always eventually
    on "n'", event []<- CSLDeliver $ "n" $ "m" }.
```

(b)

```
Inductive derives : context -> assertion -> Prop :=
| DAEvaluateP Delta Gamma Ap Ap' Ac :
  (* Replaces the head premise with its evaluation *)
  [[A Ap]] = Success Ap' ->
  Context Delta (Ap' :: Gamma) |- Ac ->
  Context Delta (Ap :: Gamma) |- Ac
  (* ... *)
| DSCut Delta Gamma Ap Ac :
  (* Add a proven assertion to the proof context *)
  Context Delta Gamma |- Ap ->
  Context Delta (Ap :: Gamma) |- Ac ->
  Context Delta Gamma |- Ac
  (* ... *)
| DPIR ctx :
  ctx |- {A: forall: "?e": event []-> "?e" =>>
    ("Fs'" $ "Fn", "Fors", "Fois") =
    request C $ "Fn" $ ("Fs" $ "Fn") $ "?e"}
```

(c)

Fig. 16. Mechanizing TLC

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 programs 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

Rajeev Alur, Kousha Etessami, and Parthasarathy Madhusudan. 2004. A temporal logic of nested calls and returns. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 467–481.

Rajeev Alur and Thomas A Henzinger. 1999. Reactive modules. *Formal methods in system design* 15, 1 (1999), 7–48.

Appendix. 2020. *Submitted Supplement Document.*

Alexander Bakst, Klaus v. Gleissenthall, Rami Gokhan Kici, and Ranjit Jhala. 2017. Verifying Distributed Programs via Canonical Sequentialization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 110 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133934

Edoardo Biagioni, Robert Harper, and Peter Lee. 2001. A network protocol stack in Standard ML. *Higher-Order and Symbolic Computation* 14, 4 (2001), 309–356.

M. Biely, P. Delgado, Z. Milosevic, and A. Schiper. 2013. Distal: A framework for implementing fault-tolerant distributed algorithms. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* 1–8. https://doi.org/10.1109/DSN.2013.6575306

Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. 2003. Deconstructing paxos. *ACM Sigact News* 34, 1 (2003), 47–67.

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. 2012. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming*. Springer, 283–307.

Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd ed.). Springer Publishing Company, Incorporated.

Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL âĂŹ14)*. Association for Computing Machinery, New York, NY, USA, 361âĂŞ372. https://doi.org/10.1145/2535838.2535881

Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. 1995. Higher-order Distributed Objects. *ACM Trans. Program. Lang. Syst.* 17, 5 (Sept. 1995), 704–739. https://doi.org/10.1145/213978.213986

Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. 2016. Formal Verification of Multi-Paxos for Distributed Consensus. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 119–136.

Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (01 Oct 2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

Bernadette Charron-Bost and André Schiper. 2009. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing* 22, 1 (01 Apr 2009), 49–71. https://doi.org/10.1007/s00446-009-0084-6

Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2010. The TLA+ Proof System: Building a Heterogeneous Verification Platform. In *Theoretical Aspects of Computing – ICTAC 2010*, Ana Cavalcanti, David Deharbe, Marie-Claude Gaudel, and Jim Woodcock (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–44.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Byron Cook, Eric Koskinen, and Moshe Vardi. 2011. Temporal property verification as a program analysis task. In *International Conference on Computer Aided Verification*. Springer, 333–348.

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Parallel Complexity Analysis with Temporal Session Types. *arXiv preprint arXiv:1804.06013* (2018).

Cezara Dragoi, Thomas A Henzinger, and Damien Zufferey. 2016. PSYNC : A partially synchronous language for fault-tolerant distributed algorithms. *Popl* (2016), 1–16. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Bruno Dutertre, Dejan Jovanovic, and Jorge A. Navas. 2018. Verification of Fault-Tolerant Protocols with Sally. In *NFM (Lecture Notes in Computer Science)*, Vol. 10811. Springer, 113–120.

Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf

Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. 2018. Paxos consensus, deconstructed and abstracted. In *European Symposium on Programming*. Springer, Cham, 912–939.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 653–669. http://dl.acm.org/citation.cfm?id=3026877.3026928

Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Peter Bodik, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. 2013. Failure Recovery: When the Cure is Worse Than the Disease. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=2490483.2490491

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. 1986. Data Refinement Refined. In *Proc. ESOP*.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

Alan Jeffrey. 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming languages meets program verification*. 49–60.

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. 2013. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Proc. FMCAD*.

Kazuhiko Kato, Atsushi Ohori, Takeo Murakami, and Takashi Masuda. 1993. Distributed C language based on a higher-order RPC technique. (1993).

Bas Ketsman, Aws Albarghouthi, and Paraschos Koutris. 2019. Distribution policies for datalog. *Theory of Computing Systems* (2019), 1–34.

Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: Language Support for Building Distributed Systems. In *Proc. PLDI*.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 234–248.

Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. 2017. A Short Counterexample Property for Safety and Liveness Verification of Fault-tolerant Distributed Algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 719–734. https://doi.org/10.1145/3009837.3009860

Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 872–923. https://doi.org/10.1145/177492.177726

Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998).

Leslie Lamport. 2000. Distributed Algorithms in TLA (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*. ACM, New York, NY, USA, 3–. https://doi.org/10.1145/343477.343497

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 357–370. https://doi.org/10.1145/2837614.2837622

Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. 2012. From Clarity to Efficiency for Distributed Algorithms. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 395–410. https://doi.org/10.1145/2384616.2384645

Nancy Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations Part I: Untimed Systems. 121 (09 1995), 214–233.

Nancy A. Lynch and Mark R. Tuttle. 1989. An introduction to input/output automata. *CWI Quarterly* 2 (1989).

Zohar Manna and Amir Pnueli. 1992. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA.

Ognjen Marić, Christoph Sprenger, and David Basin. 2017. Cutoff Bounds for Consensus Algorithms. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 217–237.

Samuel Merten, Alexander Bagnall, and Gordon Stewart. 2018. Verified Learning Without Regret. In *27th European Symposium on Programming, ESOP 2018*. 561–588.

Heather Miller, Philipp Haller, Normen Müller, and Jocelyn Boullier. 2016. Function Passing: A Model for Typed, Distributed Functional Programming. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. ACM, New York, NY, USA, 82–97. https://doi.org/10.1145/2986012.2986014

Madanlal Musuvathi and Dawson R. Engler. 2004. Model Checking Large Network Protocol Implementations. In *Proc. NSDI*.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73. https://doi.org/10.1145/2699417

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 614–630. https://doi.org/10.1145/2908080.2908118

Larry L. Peterson and Bruce S. Davie. 2003. *Computer Networks: A Systems Approach, 3rd Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Vincent Rahli. 2012. Interfacing with Proof Assistants for Domain Specific Programming Using EventML. (2012). 10th International Workshop on User Interfaces for Theorem Provers.

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 619–650.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, Washington, DC, USA, 55–74. http://dl.acm.org/citation.cfm?id=645683.664578

Guido Salvaneschi, Mirko Köhler, Daniel Sokolowski, Philipp Haller, Sebastian Erdweg, and Mira Mezini. 2019. Language-integrated privacy-aware distributed queries. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

Roopsha Samanta, Jyotirmoy V. Deshmukh, and Swarat Chaudhuri. 2013. Robustness Analysis of Networked Systems. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 229–247.

N. Schiper, V. Rahli, R. van Renesse, M. Bickford, and R.L. Constable. 2014. Developing Correctly Replicated Databases Using Formal Tools. In *Proc. DSN.*

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158116

Marcelo Taube, Giuliano Losa, Kenneth McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, , and Doug Woos. 2018. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. In *Proc. PLDI.*

Web. 2018a. Bitcoin Spinoff Hacked in Rare 51% Attack. http://fortune.com/2018/05/29/bitcoin-gold-hack/. (2018). Accessed: 2018-06-23.

Web. 2018b. Hackers have stolen about 14% of big digital currencies. http://www.latimes.com/business/la-fi-bitcoin-stolen-hackers-20180118-story.html. (2018). Accessed: 2018-06-23.

Web. 2018c. High Scalability. http://highscalability.com/blog/2011/5/2/the-updated-big-list-of-articles-on-the-amazon-outage.html. (2018). Accessed: 2018-06-23.

Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 129.

James R. Wilcox, Ilya Sergey, and Zachary Tatlock. 2017. Programming Language Abstractions for Modularly Verified Distributed Systems. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 19:1–19:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.19

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A framework for implementing and formally verifying distributed system. In *Proc. PLDI.*

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2016)*. ACM, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. 2009. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proc. NSDI.*

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proc. NSDI.*

Pamela Zave. 2012. Using Lightweight Modeling to Understand Chord. *SIGCOMM Comput. Commun. Rev.* 42, 2 (March 2012), 49–57. https://doi.org/10.1145/2185376.2185383