Trace-Based Temporal Verification for Message-Passing Programs

Jinjiang Lei and Zongyan Qiu
LMAM and Department of Informatics
School of Math, Peking University
Email: {jinjiang.lei, zyqiu}@pku.edu.cn

Zhong Shao
Department of Computer Science
Yale University
Email: zhong.shao@yale.edu

Abstract

Verification of concurrent systems is difficult because of their inherent nondeterminism. Modern verification requires clean specifications of inter-thread interferences and modular reasoning over separated components. But for message-passing models, a general reasoning system, which meets these standards, is still in demand. Here we propose a new logic for verifying distributed programs modularly. We concretize the concept of event traces to represent interactions among distributed agents, and constrain the environmental interferences by logical invariants. The verification is compositional w.r.t. agents as long as some inter-agent constraints are satisfied. Using this logic we successfully verified two classic message-passing algorithms: leader election and merging network.

1. Introduction

With the fast growth of multi-core processors and large scale distributed systems, concurrency is ubiquitous in modern software deployment. In general, agents communicate with each other via shared memory or message passing. It is difficult to verify these features because of the non-deterministic interleaving execution, and unreliable communication (e.g. message loss or duplication). Verification of shared memory concurrency has gained much progress since the emergence of Separation Logic (SL) [1], Concurrent Separation Logic (CSL) [2], [3], and other separation-based reasoning techniques [4], [5]. For message-passing models, even though they have been extensively studied in the process calculi community [6], [7], fewer general modular reasoning systems are proposed. In this paper, we develop a method based on past-tense Temporal Logic [8] to support modular verification of message-passing programs.

Unlike programs based on shared memory concurrency whose behaviors are usually confined by the well-formedness of shared data structures (e.g. linked lists, stacks), behaviors of distributed programs are mostly depicted by protocols (e.g. TCP/IP, asymmetric cryptography), which are essentially procedures used by agents to communicate with each other, e.g., “a send msg₁ to b → b send msg₂ to c → . . .”, where → represents the precedence relation among actions. This relation will be formally defined as “happens-before” later in the paper. The next action of an agent is determined by the messages that it has previously received. For this reason, past-tense Temporal Logic is suitable for formalizing the properties because of its capability to specify current state based on historical events.

In our model, the semantics of distributed systems is defined on the basis of event traces (or event graphs [9]). Event traces are Directed Acyclic Graphs (DAGs) where nodes represent atomic events, and arrows represent inter-agent message passing. The transitive closure of agents’ local order and arrows in a graph form a partial order, the “happens-before” relation, among events. In an event graph, each agent takes a local view, which is a subgraph including only casually related events with the agent, and the locality is ensured by carving out irrelevant events. On the other side, happens-before relation defines the precedence relation among events, that is specified by past-tense Temporal Logic in our method.

Technically, an agent’s local state is a pair of store and trace, \((s, tr)\): store \(s\) evaluates program variables; and trace \(tr\) is the historical record of message passing events during execution. In our model, a local trace should be closed on the happens-before relation, and therefore, it may contain events of other agents. Then the challenge is to support local reasoning while still allowing different agents to have overlapped traces. To solve this, we define a new semantics of trace conjunction, “\(tr_1 \oplus tr_2\)”, for specifying this case. The program semantics is specified in Hoare tuples. Let \(D\) be a distributed program:

\[ I \models \{p\} D \{q\} , \]

where \(I\) is a channel invariant that regulates interactions among agents, and \(p, q\) are the pre- and post-conditions that specify the local state of \(D\).

In the rest of this paper, we first formally define trace structures (Sec. 2) and its well-formedness property (Sec.3); we then define a programming language and its operational semantics (Sec. 4); we define our assertion language (Sec. 5), program logic (Sec. 6), and then apply our logic to verify a few examples in Sec. 7; finally we discuss related work and then conclude.

2. The Model

In our model, a distributed system consists of a set of agents, which communicate with each other by message passing via directed channels. Generally, agents can be threads, nodes in clusters, etc. Each agent can connect with every other agent in a system, but will not send messages to itself.

We consider asynchronous communication: send commands are non-blocking and finish immediately; receive commands...
There are several axioms that allow messages to be lost or duplicated, but do not allow them to be “overtaken” previous messages on the same channel. We use \( send(m, i) \) as the command for sending message \( m \) to agent \( i \); and use \( x := \text{recv}(i) \) for receiving a message from \( i \) and storing it in variable \( x \).

### Event Graphs

Behaviors are specified using event graphs, where each node represents an atomic send or receive event, and an arrow connects a send event to a corresponding receive event. To support asynchronous communication, we allow pending send events, but there are no pending receive events. Fig. 1 shows the trace of a distributed program, where three agents communicate with each other. Nodes are represented by ovals; \( m : i \rightarrow j \) denotes that agent \( i \) sends \( m \) to \( j \), and \( i \rightarrow j : m \) is the corresponding receive event.

In a system, an agent has its special communicating partners, therefore, it has its own local view of the event graph. The view includes the issues received by the agent and all other causally related events, which will be formally defined later. In Fig 1, the local view of agent 1 is colored by red (including \( e_1, e_2, e_3, e_4 \)); agent 2’s and 3’s are colored by blue (including \( e_1, e_2, e_3, e_5, e_6 \)) and green (only \( e_5 \)), respectively.

### Program State

Fig. 2 defines the program state. Each event consists of five components. A trace \( tr \) maps event references to events. For \( e \in \text{EventRef} \) such that \( tr(e) = (m, i, j, e', e'') \), we use \( e.val \) to denote the transmitted value \((m)\); \( e.sID \) for message’s source agent \((i)\); \( e.dID \) for the destination \((j)\); and \( e.pred \) for the local immediate predecessor \((e')\) of \( e \). We let \( e.pred = \text{nil} \) if \( e \) is the first event of an agent. If \( e \) is a receive event, \( e.send \) is its associated send event \((e'')\), otherwise \( e.send = \text{nil} \). We use \( isSend(e) \) and \( isRecv(e) \) to judge the type of \( e \).

Each agent maintains a local store and a local trace: \( \sigma = (s, tr) \). For a global state \( \Sigma = (\kappa, \theta) \), \( \kappa \) maps agent ID to local states and \( \theta \) records channel states, where \( \theta(i,j) \) is the state of the channel from agent \( i \) to \( j \), which is a queue of event traces. We will explain \( \theta \) in detail in Sec. 4. We use \( \sigma_i \) and \( \sigma_{jtr} \) to denote the components of \( \sigma \), and similar for \( \Sigma_i \) and \( \Sigma_{jtr} \). For simplicity, we assume that variable names in different agents are different, i.e., \( \text{dom}(\kappa(i)) \cap \text{dom}(\kappa(j)) = \emptyset \) when \( i \neq j \). However, traces of different agents may overlap: when \( i \neq j \), we may have \( \text{dom}(\kappa(i)tr) \cap \text{dom}(\kappa(j)tr) \neq \emptyset \).

### 3. Formal Model for Traces

Traces are special data structures for recording communications among agents during execution. In this section, we define the notion of well-formed traces.

#### Definition 1 (Happens-Before)

Let \( tr \) be a trace, the happens-before relation of \( tr \), denoted as \( \prec^{tr} \), is recursively defined as follows:

\[
\begin{align*}
& e_1 \prec^{tr} e_2 \iff e_2.\text{send} = e_1 \\
& e_1 \prec^{tr} e_2 \iff (\exists e'. e_1 \prec^{tr} e' \land e' \prec^{tr} e_2) \\
& e_1 \prec^{tr}_{loc} e_2 \iff e_1 = e_2.\text{prev} \lor (\exists e'. e_1 \prec^{tr}_{loc} e' \land e' \prec^{tr}_{loc} e_2)
\end{align*}
\]

Here \( e_1 \prec^{tr}_{loc} e_2 \) says that \( e_1 \) is either \( e_2 \)'s sender or predecessor; \( e_1 \prec^{tr} e_2 \) says \( e_1 \) happens before \( e_2 \) in \( tr \); \( e_1 \prec^{tr}_{loc} e_2 \) says additionally that \( e_1 \) and \( e_2 \) are on the same agent. We omit \( tr \) when it is obvious from the context, and use \( \preceq \) as the reflexive closure of \( \prec \).

There are five axioms to regulate the well-foundedness for traces. Let \( tr \) be a trace:

#### Axiom 1 (Self-Closed Trace)

Let \( e \in \text{dom}(tr) \), then:

- \( e.\text{send} = e' \Rightarrow e' \in \text{dom}(tr) \lor e' = \text{nil} \)
- \( e.\text{send} = e' \Rightarrow e' \in \text{dom}(tr) \lor e' = \text{nil} \)

#### Axiom 2 (Strongly Well Foundedness)

Relation \( \prec \) is strongly well-founded:

- \( \forall f : \text{dom}(tr) \rightarrow \mathbb{N}. \forall e, e' \in \text{dom}(tr). e \prec e' \Rightarrow f(e) < f(e') \)

#### Axiom 3 (Injection). Map \( e.\text{pred} \) is injective:

- \( \forall e, e' \in \text{dom}(tr). (e.\text{pred} = e'.\text{pred} \land e.\text{pred} \neq \text{nil}) \Rightarrow e = e' \)

Note \( e.\text{send} \) is not injective as messages may be duplicated.

#### Axiom 4 (Send-Receive Match)

Each receive event is matched with a send event. Let \( e \in \text{dom}(tr) \) and \( isRecv(e) \):

- \( e.\text{send} = e' \Rightarrow e.\text{val} = e'.\text{val} \land e.\text{slID} = e'.\text{slID} \land e.\text{rID} = e'.\text{rID} \land isSend(e') \)

#### Axiom 5 (FIFO-Lost-Duplicate)

Send and receive events are matched following the FIFO discipline, but messages can be lost or duplicated during transitions:

\[
\begin{align*}
& (e_1 \prec e_2 \land isRecv(e_1) \land isRecv(e_2)) \\
& \Rightarrow e_1\text{send} \leq e_2.\text{send}
\end{align*}
\]

Axiom 5 states the fact that a message cannot “overtake” another if both are transmitted via the same channel. However,
it does not ensure each message is received eventually, nor received at most once. In other words, Axiom 5 allows messages to be lost or duplicated potentially. Note that if we change the “≤” in the axiom to “<”, then messages can be lost, but not duplicated. We consider the most relaxed model, thus all theorems and rules automatically become valid for stronger models, e.g., non-lossy or non-duplicated ones. We extend the logic for stronger channels in the technical report [10].

There are some other message-passing disciplines which confine send-receive matches, e.g., S-computations, CO-computations, A-computations, etc [11]. FIFO is one of the most adopted mechanisms in many libraries, e.g., MPI.

Definition 2 (Well-Formed Trace). Trace $tr$ is well-formed, $WF(tr)$, iff it satisfies Axioms 1–5.

From now on, we consider only well-formed traces.

Trace Conjunction. We use $\text{prefix}(tr, e)$ to denote the subtrace of $tr$ stemming from $e$. The domain restriction operator $f \mid s$ restricts the domain of function $f$ within $s$.

$$\text{prefix}(tr, e) \overset{def}{=} \{ \text{e'} \mid \text{e'} \leq^\text{tr} e\}$$

If $e \notin \text{dom}(tr)$, $\text{prefix}(tr, e)$ is empty. Clearly, the empty trace is a prefix of any trace.

Lemma 1. All prefixes of a well-formed trace are also well-formed.

$$\forall e \in \text{EvtRef} : WF(tr) \Rightarrow WF(\text{prefix}(tr, e))$$

Trace structures are DAGs. Unlike trees, in general, it is impossible to separate a DAG into two well-formed sub-DAGs totally. Therefore, we use the specialized operator $tr_1 \oplus tr_2$ that allows unspecified overlaps between $tr_1$ and $tr_2$.

Definition 3 (Trace Conjunction). $tr = tr_1 \oplus tr_2$ iff the following conditions hold:

- $WF(tr) \land WF(tr_1) \land WF(tr_2)$
- $tr = tr_1 \cup tr_2$

Note that “$f \cup g$” is standard function union, which requires the operands to be consistent on overlapped domain, otherwise, it is undefined.

4. Language and Operational Semantics

Fig. 3 defines the syntax of our language. Here, send and recv are message commands; assume($B$) reduces to skip if $B$ is true, otherwise the program is blocked (when $B$ is false) or abort (when $B$ is undefined). The code of an agent can be sequential composition $C_1 \cdot C_2$, non-deterministic choice $C_1 + C_2$, and iteration $C^*$. A distributed program is a parallel composition of agents tagged with unique IDs.

Note classical control flow commands can be translated as:

- if $B$ then $C_1$ else $C_2$ $\overset{def}{=} \text{assume}(B); C_1 + \text{assume}(\neg B); C_2$
- while $B$ do $C$ $\overset{def}{=} (\text{assume}(B); C)^*; \text{assume}(\neg B)$

Growth of Trace. A trace $tr$ for a distributed system is a DAG, containing all events occurred in history. A specific agent $i$ does not know the whole graph, because some events in $tr$ may not “affect” its behavior, as shown in Fig. 1. A trace is built up by appending new events to its tail, that makes the trace growing during execution. When a send command is executed, a new send event is appended to the sender’s local trace. Due to the asynchronous communication, the message may not be received immediately, i.e., a send event is not visible by other agents until it is received. When receiving a message, the receiver must find a pending send event to match with. Once the receive event is created, its associated send event and all events that happen before the send event would be visible by the receiver. Operationally, the matched sender’s subtrace is “merged” into the receiver’s trace.

The match of send and receive events is constrained by FIFO discipline. We use structure $\theta$ as the implementation, where each $\theta(i, j)$ is a queue of sub-branches that are in transit. When agent $i$ sends $m$ to agent $j$, it appends a new send event with $m$ to its local trace, and pushes its current local trace $tr_i$ into queue $\theta(i, j)$. Note the “root” of $tr_i$ is the new send event. When agent $j$ wants to receive a message from agent $i$, it dequeues the first trace from $\theta(i, j)$ and merges the trace into its own. The dequeue is blocked if $\theta(i, j)$ is empty. Clearly, $\theta$ embodies our concept of channels.

Well-formed State. As explained, channels store partial views that help send-receive match. In order to construct well-formed traces, the state of channels should be consistent with the state of local traces. We use $\text{root}(tr)$ to denote the root of $tr$ if it stems from a single event.

$$\text{root}(tr) \overset{def}{=} \begin{cases} \text{nil} & \text{if } tr = \varnothing \\ e & \text{if there exists a unique event } e, \text{ such that } tr = \text{prefix}(tr, e) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 4 (Well-formed State). $\Sigma = (\kappa, \theta)$, where $\kappa(i) = (s_i, tr_i)$ for $i \in \{1, \ldots, n\}$, is well-formed, $WF(\Sigma)$, iff the following conditions hold:

1. There exists a trace $tr$ such that $tr = tr_1 \oplus \ldots \oplus tr_n$;
2. For each $i$, there exists a root, that is send($tr_i$), and for any event $e$ that $e \succeq_{\text{loc}} e_i$, isSend($e$) $\Rightarrow e$.sID = $i$ and isRecv($e$) $\Rightarrow e$.rID = $i$;
3. For any $i, j$, and any trace $tr$ in $\theta(i, j)$, there exists $e \in \text{dom}(tr_i)$, such that $e = \text{root}(tr)$, isSend($e$), $e$.sID = $i$ and $e$.rID = $j$;
4. For any $i, j$, if $tr_i$ is in front of $tr_2$ in $\theta(i, j)$, then $\text{root}(tr_1) \succeq_{\text{loc}} \text{root}(tr_2)$.

Condition (1) says that all the traces in $\kappa$ should be consistent on overlapped events and can be composed together to form a well-formed global trace; (2) says each local trace of an agent has a single root, and for any event $e$ on agent $i$, if $e$ is a send then $e$.sID = $i$ and if $e$ is a receive then...
During transition, we allow this to be changed autonomously.

Operational Semantics. The operational semantics is given by transition rules. Local transition takes the following form, where $i$ is agent ID, $(s, tr)$, $(s', tr')$ are local states, and $\theta, \theta'$ are states of the shared channels.

$(C, s, tr, \theta) \rightsquigarrow (C', s', tr', \theta')$

Global configuration transition takes the following form, where $\kappa$ contains the local states of all agents in $D$.

$(D, \kappa, \theta) \rightsquigarrow (D', \kappa', \theta')$

We define functions dequeue and enqueue for the operations on queues of traces. Assume a queue is represented as $tr_1 :: \ldots :: tr_n$ and let $\alpha$ be a trace.

$\begin{align*}
dequeue(tr :: \alpha) &= (tr, \alpha) \\
\text{enqueue}(\alpha, tr) &= \alpha :: tr
\end{align*}$

Semantic rules are given in Fig. 4, where $\{i_1 : v_1; \ldots ; i_n : v_n\}$ represents a function $f$ with $\text{dom}(f) = \{i_1, \ldots , i_n\}$ and $f(i) = v; f(x \parallel v)$ renames $x$ of $f$ to $v; f \uplus g$ is function union when $\text{dom}(f) \cap \text{dom}(g) = \emptyset$; $[E]_s$ and $[B]_s$ evaluate expressions according to store $s$; fresh($e$) means $e$ is a fresh event ID for the whole system. Note that the rules forbid an agent from sending (receiving) messages to (from) itself.

To mimic the case that a message may be lost or duplicated during transition, we allow $\theta$ to be changed autonomously. This is defined by the last two rules in Fig. 4.

Lemma 2. For any well-formed state $\Sigma$, and $D$, if $(D, \Sigma) \rightsquigarrow (D', \Sigma')$, then $\Sigma'$ is also well-formed.

Fig. 4. Selected Operational Semantics

5. Assertions

In this section, we introduce the assertion language to specify event traces. Fig. 5 gives the language with its semantics. Assertions emp and true specify empty traces and any well-formed trace respectively; $\text{R}(i, j, E)$ says the root of current trace is a send event where agent $i$ sends message $E$ to $j$; $\text{R}(i, j, E)$ says the root of current trace is a receive event where $j$ receives $E$ sent by $i$. For the connectors, $\text{p} \lor \text{q}$ holds if the trace stemming from the predecessor of current root satisfies $p$; $\text{p} \land \text{q}$ holds if the trace stemming from the sender field of current root satisfies $q$; $p \lor q$ holds over $tr$ if there is a path tracing back from the root of $tr$ and an event $e$ in the path such that $p$ holds over prefix$(tr, e)$ and $q$ holds since then; $p \land q$ holds over $tr$ if for all paths that tracing back from the root of $tr$, there is an event $e$ in the path that $p$ holds over prefix$(tr, e)$ and $q$ holds since then; $p \lor q$ lifts separating conjunction to traces.

Other useful connectors can be defined using the primitives: $p \lor q$ is a weaker version of $p \lor q$; assertion $p$ says $p$ was once true in the history; $\parallel p$ says $p$ holds everywhere; $p \lor q$ is a weaker form of $p \lor q$; $p$ says for any path starting from the current root, there exists an event in the path where $p$ holds; and $\parallel p$ says there exists a path starting from the root such that every event in the path entails $p$.

6. The Logic

Now we present our logic for trace reasoning. We use Hoare tuple $I \vdash \{p\} D \{q\}$ and $I \vdash \{p\} C \{q\}$ for program $D$ and $C$, respectively, where $\vdash$ is a special form of $\vdash$:

$I \vdash \{p\} C \{q\} \equiv I \vdash \{p\} i : C \{q\}$
traces in a channel. Intuitively, invariant \( I \) is an ordinary channel invariant. Other rules are standard. Let \( n \) be a fresh variable that satisfies an invariant \( I \). According to Def. 5, \( I \) contains no free variables, therefore, the state of store is irrelevant in Def. 6.

We list a set of inference rules in Fig. 6. Rule (RECV) specifies receive commands. In the postcondition, \( \circ_{\text{and}} r \) means the local agent will pull a trace from the channel which is guaranteed to satisfy invariant \( I(j,i) \); and the pre-assertion \( p \) becomes \( \circ_{\text{pred}} \) in the post. Rule (SEND) specifies send commands. The sender must ensure the new trace satisfies the channel invariant. Other rules are standard.

Due to page limit, we leave the semantic definition and soundness proof in the technical report [10].

7. Examples

In this section, we give the proofs for two examples.

Leader Election. Assume there are \( n \) agents that are connected in a ring. We count them mod \( n \), then 0 is another name for agent \( n \), \( n + 1 \) is another name for agent 1, etc. Each agent \( i \) holds a unique fixed positive integer token, and a local boolean variable \( ld_i \) whose initial value is false. When the program terminates, the agent who has the biggest token wins, and its local variable \( ld_i \) is set to true.

The program is given in Fig. 7. Each agent sends its token following the ring at the start and then goes into a loop. When an agent receives an incoming token, it compares the token with its own. If the incoming token is greater, it keeps passing the token; if the token is less, it discards the incoming message by doing nothing; if it is equal to its own, the agent sends out
agent_1():
    ld_1 := ff;
    send ((token, i+1);

agent_1():
    {emp}
    loop \def
def
    assume (tk_i = 0);
    send (token, i+1);
    \{assume (tk_i < token)};

Fig. 7. Leader Election: Program

0 to claim the leader has been chosen, and all agents terminate
after 0 has past around the ring.

We use maxtk(i, j) to represent the largest token from agent
i to j (including both i and j). Assertion p(i) says if agent i
sends token_j to the next, then token_j is the largest token from
j to i:

p(i) \def \forall j : S(i, i+1, token_j)
I(i, i+1) \def p(i)

Note that token_j is not a variable, but is a predefined constant,
thus all maxtk(i, j) are constants as well.

We are expected to prove that if the local variable ld_i =
tt, then agent i holds the largest token around the ring. This
property is specified by the following tuple:

I \vdash \{emp\} agent_i() \{ld_i = tt \Rightarrow token_i = maxtk(1, n}\)

The proof sketch is given in Fig. 8. Send and receive commands follow (SEND) and (RECV) rules in Fig. 6: the pre-
state p of a send becomes \circ pred p in the post-state; and for a
receive, it additionally pulls the channel invariant p(i-1) as
\circ send p(i-1) in its post-state. The proof of loops is presented
separately in Fig. 8, which ensures every message sent to the
next agent satisfies channel invariant and the loop invariant
remains valid at the end of the loop. The code after the loop
just ensures the termination of the system.

According to rule (PAR), let q_i \def \{ld_i = tt \Rightarrow token_i = maxtk(1, n)\},
the specification of the overall system is:

\{emp\}
agent_1() || ... || agent_n()
{q_1}
{q_1 \oplus ... \oplus q_n}

According to the post condition: if the largest token is unique,
then there is at most one ld_i whose value is tt.

Note that the proof is still valid when tokens are lost or
duplicated. If the largest token is lost, no agent can be the
leader and the system would not terminate; if other tokens are
lost, it does not affect the result. On the other hand, it is also
trivial that duplicated tokens do not affect the result, even if
the largest token is duplicated.

Leader election has been proved by many others, e.g. logic
of events [12]. However, there are three major differences
that distinguish our logic: (1) Our proof is modular. Each
agent is proved by its reliance of its incoming channel and
guarantee to its out going channel. Therefore, different agents
may be implemented differently as long as they respect the
channel invariants. (2) Most other works rely on mathematical
abstractions and auxiliary lemmas, while ours make directly
deduction over the source code. (3) Moreover, our proof is
still valid under unreliable message passing.

Merging Network. Filters is a common kind of distributed
systems. A filter is an agent that receives messages from some
input channels and sends messages to some output channels.
Here we prove an example of filters — Merging Network.
Fig. 9 (upper) shows the structure of a merging network. Each
agent here merges two monotonic increasing positive integer
streams into one increasing stream, where 0 marks the end of
Fig. 9. Merge Sort: Agent Struct. and Code of Agent 5

streams. Fig. 9 gives the algorithm of agent 5 in the network: it receives streams from agent 3 and 4, merges both streams and sends out the merged result to agent 6.

The interface between agents is defined by an invariant:

\[ I(sndr,rcvr) \triangleq \Diamond S(sndr,rcvr, n_2) \land S(sndr,rcvr, n_1) \Rightarrow 0 < n_2 \leq n_1 \land n_1 = 0 \]

Where \( sndr \) and \( rcvr \) are agent IDs, and \( I(sndr,rcvr) \) specifies that in the channel from \( sndr \) to \( rcvr \), the current message is no less than any previous messages except the current message is 0.

The correctness of agent 5 is specified by tuple:

\[ I \vdash \{ \text{emp} \} \text{agent5}() \{ \text{monoS}(5,6) \} \]

where \( \text{monoS}(i,j) \triangleq \neg \Diamond S(i,j, n_2) \land S(i,j, n_1) \Rightarrow 0 < n_2 \leq n_1 \land n_1 = 0 \)

The post-condition \( \text{monoS}(5,6) \) says that agent 5 sends an increasing positive number stream to agent 6 until the end-mark 0: take any \((\not\Diamond)\) event in the trace of agent 5, if this event sends \( n_1 \) to agent 6 and an earlier \((\Diamond)\) event sends \( n_2 \) to agent 6, then either \( n_2 \leq n_1 \) or \( n_1 \) is the end mark \((n_1 = 0)\). Here \( n_1 \) and \( n_2 \) are implicitly quantified by for all \((\forall)\).

Fig. 10 gives the proof sketch, with some invariants:

\[ \begin{align*}
I_{v_1} & \triangleq \text{monoS}(5,6) \\
I_{v_2} & \triangleq \forall n \cdot \Diamond S(5,6,n) \Rightarrow (v_1 = 0 \lor v_1 \geq n) \land (v_2 = 0 \lor v_2 \geq n) \\
I_{v_3} & \triangleq \exists n_1, n_2 \cdot \Diamond S(3,5,n_1) \land v_1 = n_1 \land \Diamond S(4,5,n_2) \land v_2 = n_2 \\
I & \triangleq I_{v_1} \land I_{v_2} \land I_{v_3}
\end{align*} \]

\( I_{v_1} \) says agent 5 sends monotonic streams to agent 6; \( I_{v_2} \) says variables \( v_1 \) and \( v_2 \) are either equal to 0 or no less than any previous message had sent; \( I_{v_3} \) says \( v_1 \) and \( v_2 \) always equal to a value which was previously received from agent 3 and 4, respectively.

In the system aspect, the algorithm in Fig. 9 shows one verified implementation of agent 5. It is obviously feasible that other agents may adopt different algorithms, as long as they ensure the channel invariants. Since agent interferences are neatly confined by channels invariants, implementations of other agents would not affect the existing proof in Fig. 10.

This proof is also valid through message loss and duplication. Note that message loss and duplication do not affect the FIFO property of the message sequence, therefore, the stream is still monotonic, and invariants hold everywhere.

8. Related Work and Conclusions

Verification of message-passing programs has been studied from various standpoints for decades. There exist many famous process calculi, e.g., CSP (Communicating Sequential Processes) [6], and CCS (Calculus of Communicating System) [13]. However, those algebraic systems focus mainly on agent behavior deductions and equivalence, e.g., bi-simulation. It is unclear how to apply those calculi to modularly specify and reason the properties of local computing for actual code of agents with states, that is our focus in this work. Thus, we will compare our approach mainly with recent work on modular program verification.

Concurrent Separation Logic. The specification of a program is inspired by CSL [2], [3], which uses a predefined \( I \) as the
invariant of shared resource. In this case, even though local traces of different agents may overlap, rather than viewing those shared (overlapped) events as shared resource, we treat channels as shared resource. This makes it easier to specify inter-agent interferences.

In addition, separating conjunction (∗) has its root from SL where it is used to support heap separation. However, unlike heaps in SL, the states of traces are well-structured graphs with many add-on restrictions. Therefore, we redefine the semantics of ∗ to make it a reasonable trace separation operator.

**Trace Semantics.** Lamport event graph [9] is one of the earliest state-based semantics for distributed programs. To support better modularity, we decomposed the global event graph into separate subgraphs (for each agent), and used channels to transmit such subgraphs among agents. This way, the states of different agents are isolated, and interferences among agents can be neatly confined by the state of channels.

**Temporal Logic.** Temporal logic is widely used in program specification. Roever et al. [14] presents a good overview of recent work on verification of concurrent programs. The leading theme of the book is compositional techniques for concurrency verification, and clearly, our work can be viewed as a new candidate along this direction. The book introduced a type of temporal logic for message-passing models as well. The trace semantics defined there is a sequence of states transition σ1 :: σ2 :: ... , where each σi is a snapshot of the system state. Our trace definition is different: all temporal assertions specify the same data structure at different times; and each node in the trace is either a send or a receive event but not the whole program state.

**State-based Reasoning.** State-based reasoning is clearly related to Hoare-style semantics, where “state” could be heap as in SL, or “trace” as we do, or any other data structure.

Based on the state of heap, Villard et al. [15] proposed a logic for copyless message passing and implemented it in a tool called Heap-Hop [16]. They support ownership transfer, and interactions are specified by finite automats. Villard’s method is defined based on shared memory model such that copyless message passing is feasible, but ours is for pure message-passing systems. Our method is extensible for ownership transfer as well: one solution is to add “resource” (e.g. heap) for each agent, and specify those transmitted resources inside channel invariants. However, this solution is similar to CSL. It may complicate our framework.

Bell et al. [17] defined the state of a channel as a queue of messages, and, based on that, proposed a variant of CSL for reasoning about message-passing programs; they also used channel invariants to specify agent interactions as we do. However, if two agents use the same channel, they would merge their local records to get a set of all possible interleavings. Our method considers a more abstract setting where we need not to consider those annoying interleavings. Our trace invariants are more general because they are trace assertions, thus are not limited to specify a particular channel; instead, they can specify the overall behavior of agents.

Others. We list some other related works here: Calcagno et al. [18] use trace-based semantics for reasoning about shared memory concurrency. Bickford et al. [19] formally define the event trace structures, and give a minimal set of axioms for trace reasoning. Francalanza et al. [20] propose permission-based logic for message-passing concurrency.

**Conclusion and Future Work.** In this paper we developed a trace semantics and a program logic for verifying the functional correctness of message-passing (distributed) programs. We showed the feasibility that distributed program verification can be local and modular. By combining with trace semantics and temporal logic, we presented a natural way for specifying protocols. We presented in the paper how to apply our logic to verify two message-passing programs. In the future, we would like to further test its applicability with more applications and build tools to automate the verification process.

**References**


