Trace-Based Temporal Verification for Message-Passing Programs
(Technical Report)

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 usually 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.

Supported by NNSF of China, Grant No. 61272160, 61100061, and 61202069
ners, therefore, it has its own local view of pending send events, but there are no pending receive events.

In Fig 1, agent 1's local view shows the graph of messages sent from other agents. For instance, agent 1 sends a message to agent 3.

Fig. 1. Event Graph Example

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 are blocked when the designated channel is empty. For generality, we allow messages to be lost or duplicated, but do not allow a message to “overtake” previous messages on the same channel. We use send(m, i) as the command for sending message m to agent i; and use x := 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 → j denotes that agent i sends m to j, and i → 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 events issued 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 e1, e2, e3, e4); agent 2’s and 3’s are colored by blue (including e1, e2, e3, e5, e6) and green (only e5), 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 ∈ 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.rID for the destination (j); and e.pred for the local immediate predecessor (e′) of e. We let e.pred = 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 = nil. We use isSend(e) and isRecv(e) to judge the type of e.

Each agent maintains a local store and a local trace: σ = (s, tr). For a global state Σ = (κ, tr), κ maps agent ID to local states and tr records channel states, where θ(i, j) is the state of the channel from agent i to j, which is a queue of event traces. We will explain θ in detail in Sec. 4. We use σi and σtr to denote the components of σ, and similar for Σκ and Σθ. For simplicity, we assume that variable names in different agents are different, i.e., dom(κ(i)) ∩ dom(κ(j)) = ∅ when i ≠ j. However, traces of different agents may overlap: when i ≠ j, we may have dom(κ(i)tr) ∩ dom(κ(j)tr) ≠ ∅.

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, ≺tr, is recursively defined as follows:

\[ e_1 ≺_{tr} e_2 \quad \text{by} \quad e_1.\text{send} = e_2.\text{send} \]
\[ e_1 ≺_{tr} e_2 \quad \text{by} \quad e_1.\text{send} \preceq e_2.\text{send} \]
\[ e_1 ≺_{tr} e_2 \quad \text{by} \quad e_1.\text{send} = e_2.\text{send} \]

Here e1 ≺tr e2 says that e1 is either e2’s sender or predecessor; e1 ≺tr e2 says e1 happens before e2 in tr; e1 ≺tr e2 says additionally that e1 and e2 are on the same agent. We omit tr when it is obvious from the context, and use ≤ as the reflexive closure of ≺.

There are five axioms to regulate the well-formedness for traces. Let tr be a trace:

Axiom 1 (Self-Closed Trace). Let e ∈ dom tr, then:

- e.\text{pred} = e’ ⇒ e’ ∈ dom(tr) \land e’ ≠ nil
- e.\text{send} = e’ ⇒ e’ ∈ dom(tr) \land e’ ≠ nil

Axiom 2 (Strongly Well Founded). Relation ≺ is strongly well founded:

- \( \exists f : \text{dom}(tr) \rightarrow \text{Nat} \cdot \forall e, e’ \in \text{dom}(tr) \cdot e \prec e’ \Rightarrow f(e) < f(e’) \)

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

- \( \forall e, e’ \in \text{dom}(tr) \cdot (e.\text{pred} = e’.\text{pred}) \Rightarrow e = e’ \)

Note •.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 ∈ dom tr and i isRecv(e):

- e.\text{send} = e’ ⇒ e.\text{val} = e’.\text{val} \land e.\text{sID} = e’.\text{sID} \land e.\text{rID} = e’.\text{rID} \land \text{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:

\[ \begin{align*}
\text{if} & \quad e_1 \prec e_2 \land \text{isRecv}(e_1) \land \text{isRecv}(e_2) \\
\text{then} & \quad e_1, \text{siD} = e_2, \text{siD} \land e_1, \text{riD} = e_2, \text{riD}
\end{align*} \]

\[ \Rightarrow e_1, \text{send} \preceq e_2, \text{send} \]

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 “\(\preceq\)” in the axiom to “\(\succeq\)”, 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 Appendix.

There are some other message-passing disciplines which confine send-receive matches, e.g., S-computations, CO-computations, A-computations, etc [10]. 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 prefix \((tr,e)\) to denote the subtrace of \(tr\) stemming from \(e\). The domain restriction operator \(f | s\) restricts the domain of function \(f\) within \(s\).

\[ \text{prefix}(tr,e) = \{ e' \mid e' \succeq^{tr} e \} \]

If \(e \notin \text{dom}(tr)\), 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.

\[ \forall e \in \text{EventRef}. \quad WF(tr) \Rightarrow WF(\text{prefix}(tr,e)) \]

**Proof:** Let \(tr\) a trace, and \(e \in \text{dom}(tr)\), we need to prove prefix \((tr,e)\) is well-formed.

- **Axiom 1** Because \(tr\) is self-closed, and prefix \((tr,e)\) includes all events that happen before \(e\), therefore, prefix \((tr,e)\) is self-closed.
- **Axiom 2** Assume \(f\) is the function for \(tr\) that let Axiom 2, we define \(f' = f \setminus \{ e \in \text{dom}(\text{prefix}(tr,e)) \}\), then \(f'\) demonstrates the validate of Axiom 2 for prefix \((tr,e)\).
- **Axiom 3** Because prefix \((tr,e)\) is a subtrace of \(tr\), and Axiom 3 holds for \(tr\), the axiom holds for prefix \((tr,e)\) either.
- **Axiom 4** The axioms holds trivial for prefix \((tr,e)\).
- **Axiom ??** Because prefix \((tr,e)\) is a subtract of \(tr\), if the axiom doesn’t hold for prefix \((tr,e)\), the axiom is not valid for \(tr\) either.

**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:

\[ \begin{align*}
\text{(AgentID) } i, j & : \text{Nat} \\
\text{(Expr)} E & ::= x \mid m \mid E + E \mid E - E \mid \ldots \\
\text{(BExp)} B & ::= \text{true} | \text{false} | E = E | E \neq E | \ldots \\
\text{(Comd)} c & ::= x ::= E \mid \text{skip} \mid \text{assume}(B) \mid \text{send}(E,i,i) \mid x ::= \text{recv}(i) \\
\text{(Stmts)} C & ::= e_1 \lor e_2 \lor C_1 \land C_2 \lor C_1 + C_2 \lor C \lor C_1 \land \text{recv}(i) \lor C_2 \\
\text{(Prog)} D & ::= i : C \mid D \lor D
\end{align*} \]

Fig. 3. The Language

- **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 passing 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 \(C_1 \land C_2\), 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:

\[ \begin{align*}
& \text{if} B \text{ then } C_1 \text{ else } C_2 \\& \text{ do } C \text{ while } B \text{ do } C \equiv \text{assume}(B) \land C_1 + \text{assume}(\neg B) \land C_2 \land \text{assume}(\neg B)
\end{align*} \]

**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 subtraces 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 root(tr) to denote the root of tr if it stems from a single event.

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

**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 \( tr = tr_1 \uplus \ldots \uplus tr_n \);
2. For each \( i \), there exists a root, that is \( e_i = \text{root}(tr_i) \), and for any event \( e \) that \( e \preceq \text{loc } e_i \), isSend(e) \( \Rightarrow e.\text{sID} = i \) 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 \( e = \text{root}(tr_i) \), isSend(e) \( \Rightarrow e.\text{sID} = i \) and \( e.rID = j \);
4. For any \( i, j \), if \( tr_1 \) is in front of \( tr_2 \) in \( \theta(i, j) \), then root(tr1) \( \preceq \text{loc } \text{root}(tr_2) \);

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 \( e.\text{sID} = i \) and if \( e \) is a receive then \( e.rID = i \); (3) says each trace in a channel \( \theta(i, j) \) has one root which must be a send event from agent \( i \) to \( j \); (4) says the sequence of traces in a channel should be consistent with the happens-before relation in the graph.

**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.

\[
\begin{align*}
&\text{local transition rules:} \\
&(1) (s, tr) \Rightarrow (s', tr') \text{ by transition rules.}
\end{align*}
\]

\[
\begin{align*}
&(2) (s, tr) \Rightarrow (s, tr') \text{ if } tr' = tr \cup \{e' : (n, i, j, e, nil) \} \text{ for } (i, j) \notin \text{dom}(\theta) \text{ or } |E| \text{ undefined}
\end{align*}
\]

\[
\begin{align*}
&(3) (s, tr) \Rightarrow (s, tr') \text{ if } tr' = \text{dequeue}(\theta(i, j), tr) \text{ for the operations dequeue, enqueue}
\end{align*}
\]

\[
\begin{align*}
&(4) (s, tr) \Rightarrow (s, tr') \text{ if } tr' = \text{send}(\theta(i, j), tr, \{i(i) \mapsto \mu\}) \text{ for the operations send, receive}
\end{align*}
\]

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

\[
\begin{align*}
&(D, \kappa, \theta) \Rightarrow (D', \kappa', \theta') \text{ if } \theta(j, i) = \alpha_1 :: tr :: \alpha_2 \text{ and } \theta(j, i) = \theta(j, i) \mapsto \alpha_1 :: tr :: \alpha_2
\end{align*}
\]

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:

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

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 \( f(i_j) = v_j \). \( \text{remaps } x \text{ to } v \) if \( f \) is a function union when \( \text{dom}(f) \cap \text{dom}(g) = \emptyset \). \( |E| \text{ and } [B] \) 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 \( (D, \Sigma) \Rightarrow (D', \Sigma') \), then \( \Sigma' \) is also well-formed.

**Proof:** The lemma is proved by induction over the operational rules.

If the step is caused by a send command of thread \( i \) and send messages to \( j \), (1) still holds trivially; (2) and (3) holds because the added event \( e \) is a send event at the end of \( tr'_j \), with \( e.\text{sID} = i \) and \( e.rID = j \); (4) and (5) holds because the
added trace corresponding with the last send event to thread \(j\).

If the step is caused by a receive command of thread \(i\) that receive message from \(j\), let \(tr'_i\) is the post local trace of thread \(i\), and \(tr\)' the post global trace. (1) we need to prove \(tr\) and \(tr'_i\) are both well-formed. Axiom 1 – Axiom 4 holds trivially. Axiom ?? holds because the receive command matched with the first trace in the channel, and the channel satisfy (3) – (5). The receive command therefore matched with the first unmatched send event form \(j\) to \(i\). (2) holds trivially. (3) – (5) holds because receive commands does not add more traces into any channels.

\[\Box\]

5. Assertions

In this section, we introduce the assertion language to specify events. Fig. 5 gives the language with its semantics. Assertions emp and true specify empty traces and any well-formed trace respectively: \(S(i, j, E)\) says the root of current trace is a send event where agent \(i\) sends message \(E\) to \(j\); \(R(i, j, E)\) says the root of current trace is a receive event where \(j\) receives \(E\) sent by \(i\). For the connectors, \(\circ\text{pred}\) \(p\) holds if the trace stemming from the predecessor of current root satisfies \(p\); \(\circ\text{snd}\) \(p\) holds if the trace stemming from the sender field of current root satisfies \(q\); \(p \triangleright q\) 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 \(\text{prefix}(tr, e)\) and \(q\) holds since then; \(p \triangleright q\) 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 \(\text{prefix}(tr, e)\) and \(q\) holds since then; \(p \circ q\) lifts separating conjunction to traces.

Other useful connectors can be defined using the primitives: \(p \triangleright q\) is a weaker version of \(p \triangleright q\; \text{assertion}\; \circ \; p\) says \(p\) was once true in the history; \(\circ\) \(p\) says \(p\) holds everywhere; \(p \triangleright q\) is a weaker form of \(p \triangleright q\; \text{▷} \; p\) says for any path starting from the current root, there exists an event \(e\) in the path where \(p\) holds; and \(\Box\) \(p\) says there is 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 \(I \vdash_i \{p\} C \{q\}\) for program \(D\) and \(C\), respectively, where \(\vdash_i\) is a special form of \(\vdash\):

\[I \vdash_i \{p\} C \{q\} \equiv I \vdash \{p\} i : C \{q\}\]

Invariant \(I\) specifies the shared resource (channels) of agents with the type:

\[I : \text{AgntID} \times \text{AgntID} \rightarrow \text{fin\; Assertion}\]

Intuitively, invariant \(I(i, j) = p\) requires every trace transmitted via \(\theta(i, j)\) satisfies \(p\). Note that \(I(i, j)\) is an ordinary trace assertion: it specifies properties for each individual trace in \(\theta(i, j)\), but does not specify the relation between different traces in a channel.

**Definition 5 (Well-defined Invariant).** Let \(I\) be an invariant for channels, \(I\) is well-defined iff for all \((i, j) \in \text{dom}(\theta)\), \(\text{freeVar}(I(i, j)) = \emptyset\).

Def. 5 is proposed for the consideration of soundness, where \(\text{freeVar}(p)\) returns the free variables of an assertion. Since the states of stores are not shared, the validity of channel state should not rely on any local variable. From now on, we consider well-formed invariants only.

**Definition 6.** \(\theta \models I\) holds, iff for all \((i, j) \in \text{dom}(\theta)\) and for all \(tr\) in queue \(\theta(i, j), \langle - , tr \rangle \models I(i, j)\), where \(\langle - \rangle\) represents a store whose value is irrelevant.

Def. 6 defines the case when a channel \(\theta\) 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 \((\text{RECV})\) specifies receive commands. In the postcondition, \(\circ\text{snd}\) \(r\) means the local agent will pull a trace from the channel which is
An agent receives an incoming token, it compares the token following the ring at the start and then goes into a loop. When the program terminates, the agent who has the biggest tokenld a local boolean variable is set to true.

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 tokenj to the next, then tokenj is the largest token from j to i:

\[ p(i) \triangleq \forall j \cdot S(i, i + 1, \text{token}_j) \Rightarrow \text{token}_j = \text{maxtk}(j, i) \]

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

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

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 tokeni 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 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:

\[ p(i) \triangleq \forall j \cdot S(i, i + 1, \text{token}_j) \Rightarrow \text{token}_j = \text{maxtk}(j, i) \]

Note that token_i is not a variable, but is a predefined constant, thus all maxtk(j, i) 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:

\[ I \vdash \{ \text{emp} \} \text{agent}_i() \{ ld_i = \text{tt} \Rightarrow \text{token}_i = \text{maxtk}(1, n) \} \]

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 ⊗_pred p in the post-state; and for a receive, it additionally pulls the channel invariant p(i − 1) as ⊗_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 \( q_i \triangleq \{ ld_i = \text{tt} \Rightarrow \text{token}_i = \text{maxtk}(1, n) \} \), the specification of the overall system is:

\[
\{ \text{emp} \} \quad \text{agent}_1() \quad || \quad \ldots \quad || \quad \text{agent}_n()
\]

\[
\{ q_1 \} \quad || \quad \ldots \quad || \quad \{ q_n \}
\]

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 [11]. 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 outgoing 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
agent();
{
  {emp}
  ld_i := ff;
  {emp ∧ ld_i = ff}
  send (token, i+1); 
  {ld_i = ff ∧ S(i, i+1, token_{i}) ∧ ⊓pred emp}
  {ld_i = ff ∧ p(i)}
  tk_i := recv (i-1);
  {ld_i = ff ∧ R(i-1, i, tk_i) ∧ ⊓pred p(i) ∧ ⊓and p(i-1)}
  {R(i-1, i, tk_i) ∧ ⊓pred p(i) ∧ ⊓and p(i-1) ∧
  (ld_i = tt ⇒ token_{i} = maxtk(1, n))}
  loop^* (see below)
  {md_{i, i+1} := recv (i-1);} 
  {md_{i, i+1} = recv (i-1)}
  {md_{i, i+1} = recv (i-1)}
  {md_{i, i+1} = recv (i-1)}
  {md_{i, i+1} = recv (i-1)}
  send (0, i+1); 
  {md_{i, i+1} = recv (i-1)}
  {md_{i, i+1} = recv (i-1)}
  {md_{i, i+1} = recv (i-1)}
  {md_{i, i+1} = recv (i-1)}
  loop:
  {R(i-1, i, tk_i) ∧ ⊓pred p(i) ∧ ⊓and p(i-1)}
  {ld_i = tt ⇒ token_{i} = maxtk(1, n)}
  assume (tk_{i} = 0);
  {ld_i = tt ⇒ token_{i} = maxtk(1, n)}
  + {assume (ld_i = ff); send (0, i+1);}
  {ld_i = tt ⇒ token_{i} = maxtk(1, n)}
  }

Fig. 8. Leader Election: Proof

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 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) ≡ ⊓S(sndr, rcvr, n_2) ∧ S(sndr, rcvr, n_1)
⇒ 0 < n_2 ≤ n_1 ∨ 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 ⊢ {emp} agent5 () {monoS(5, 6)}
where monoS(i, j) ≡
(∃S(i, j, n_2) ∧ S(i, j, n_1) ⇒ 0 ≤ n_2 ≤ n_1 ∧ n_1 = 0)

The post-condition monoS(5, 6) says that agent 5 sends an increasing positive number stream to agent 6 until the endmark 0: take any (⇒) event in the trace of agent 5, if this event sends n_1 to agent 6 and an earlier (⇒) event sends n_2 to agent 6, then either n_2 ≤ n_1 or n_1 is the end mark (n_1 = 0). Here n_1 and n_2 are implicitly quantified by for all (∀).

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

Inv_1 ≡ monoS(5, 6)
Inv_2 ≡ Vn. ⊓S(5, 6, n) ⇒ (v_1 = 0 ∨ v_1 ≥ n) ∧ (v_2 = 0 ∨ v_2 ≥ n)
Inv_3 ≡ Vn_1, n_2. ⊓S(3, 5, n_1) ∧ v_1 = n_1 ∧ ⊓S(4, 5, n_2) ∧ v_2 = n_2

Inv_1 says agent 5 sends monotonic streams to agent 6; Inv_2 says variables v_1 and v_2 are either equal to 0 or no less than any previous message had sent; Inv_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
agent5 ()
{
    (emp)
    \{v_1 = recv 3; v_2 = recv 4\}
    \{ \ominus \text{pred} \ominus \text{pred} \ominus \text{emp} \land \ominus \text{pred} \ominus \text{R}(3,5,v_1) \land \ominus \text{R}(3,5,v_2) \}\n    \{Inv\}
    loop \{ (\text{see below}) \}
    \{Inv\}
    \{assume (v_1 = 0); \}
    \{Inv \land v_1 = 0\}
    \{assume (v_2 \neq 0); send \{v_2, 6\}; v_2 = recv 4; \}
    \{Inv \land v_1 = v_2 = 0\}
    \{\text{send} \{0, 6\}; \}
    \{Inv\}
    \{\text{mono}5(5, 6)\}
    loop:
    \{Inv\}
    \{assume (v_1 > v_2 > 0); \}
    \{Inv \land v_1 > v_2 > 0\}
    \{\text{send} \{v_2, 6\}; \}
    \{\ominus S(5,6,v_2) \land \ominus \text{pred} \ominus \text{Inv} \land v_1 > v_2 > 0\}
    \{\exists n_2 \cdot \ominus S(4,5,n_2) \land S(5,6,n_2) \land \ominus \text{pred} \ominus \text{Inv} \land v_1 > n_2 > 0\}
    \{v_2 = recv 4; \}
    \{\ominus R(4,5,v_2) \land \ominus \text{pred} \ominus \text{I}(4,5) \land \exists n_2 \cdot \ominus S(4,5,n_2) \}
    \{\ominus \text{pred} \ominus S(5,6,n_2) \land \ominus \text{pred} \ominus \text{pred} \ominus \text{Inv} \land v_1 > n_2 > 0\}
    \{Inv\}
    \{assume (0 < v_1 \leq v_2); send \{v_1, 6\}; v_1 = recv 3; \}
}

Fig. 10. Merge Sort: Proof

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) [12]. 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 ($\ast$) 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 $\ast$ 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. [13] 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 $\sigma_1::\sigma_2::\ldots$, where each $\sigma_i$ 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. [14] proposed a logic for copyless message passing and implemented it in a tool called Heap-Hop [15]. They support ownership transfer, and interactions are specified by finite automata. 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. [16] 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. [17] use trace-based semantics for reasoning about shared memory concurrency. Bickford et al. [18] formally define the event trace structures, and give a minimal set of axioms for trace reasoning. Fracalanza et al. [19] 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

Appendix

In this appendix, we show the soundness of our logic.

Axiom 5 allows messages to be lost or duplicated during transmission. In this section, we present the strict FIFO transmission, which forbids both.

Axiom 5' (FIFO). Send and receive events were matched following the FIFO message passing discipline. Let $e_1, e_2$ be any events in $tr$ that satisfy $\text{isSend}(e_1)$ and $\text{isSend}(e_2)$:

- $e_1 \prec e_2$ and $e_1, \text{slID} = e_2, \text{slID} \land e_1, \text{rID} = e_2, \text{rID} \land \ \text{received}(e_2) \Rightarrow \text{received}(e_1)$

where $\text{received}(e) \overset{\text{def}}{=} \text{isSend}(e) \land \exists e' \in \text{dom}(tr). e'. \text{send} = e$.

Axiom 5' ensures messages are not lost. To ensure non-duplication, we need to change $\leq$ in Axiom 5 to $\prec$ as well.

We consider strict FIFO message-passing in this section.

Send-Receive Match. One key issue for message passing programs is matching, that is how send and receive events were paired. Only by matching a receive event with its sender, we can deduce the specific value that the receive event associates with. On the other hand, as we will show later, send-receive matching helps to clarify the structure of traces, rather than unspecified overlapping specified by $\otimes$.

As mentioned in Sec. 3, our message passing follows FIFO discipline. In the operational semantics, we use channels to help pairing receive events with dangling send events. We can also solve this problem by introducing channel assertions for specifying states of channels, but this is an ad hoc solution that depends on specific implementations. With Temporal Logic, we can use our assertion to deduce send-receive match directly.

Definition 7. $\kappa \models p_i \otimes_j q$ iff $\kappa(i) \models p$ and $\kappa(j) \models q$ and let $tr = \kappa(i)_t \otimes \kappa(j)_t r$, for any $e \in \text{dom}(tr)$, if $\text{isSend}(e) \land e, \text{slID} = i \land e, \text{rID} = j$, then there exists $e' \in \text{dom}(tr)$, that $e'. \text{send} = e$.

The following rules are sound based on Def. 7.

$$\begin{align*}
\text{emp}_i \otimes_j \text{emp} & \quad (S(i, j, -) \land \ominus \text{pred} p) i \otimes_j (R(i, j, -) \land \ominus \text{pred} q) \\
p_i \otimes_j q & \quad j' \neq j \\
(\neg S(i, j', -) \land \ominus \text{pred} p) i \otimes_j q & \quad p_i \otimes_j (R(j', j, -) \land \ominus \text{pred} q)
\end{align*}$$

An important benefit obtained from the matching is that we can now deduce precise trace structures from unspecified trace overlapping, as shown by following rules.

$$\begin{align*}
n_i \otimes_j p & \Rightarrow (S(i, j, -) \land \ominus \text{send} q) \\
& (p \otimes q) \Rightarrow p \land \ominus \text{send} q \\
q_i \otimes_j p & \Rightarrow (S(i, j, -) \land \ominus \text{send} q) \\
& (p \oplus q) \Rightarrow (S(i, j, -) \land \ominus \text{send} q) \land (C_1^{\text{ctx}}[p] \lor C_2^{\text{ctx}}[q]) \\
& (p \lor \ominus \text{send} q) \Rightarrow (S(i, j, -) \land \ominus \text{send} q) \land (C_1^{\text{ctx}}[p] \lor C_2^{\text{ctx}}[q])
\end{align*}$$

Where $C^{\text{ctx}}[q]$ is an assertion which has a sub-formula $p$, and the context is specified by $C^{\text{ctx}}$. Once a receive is matched with a send, the value must be matched.

$$R(i, j, E) \Rightarrow \ominus \text{send} S(i, j, E)$$

Proof of Lemma 1.

Proof: Let $tr$ be a trace, and $e \in \text{dom}(tr)$, we need to prove $\text{prefix}(tr, e)$ is well-formed.

- Axiom 1. Because $tr$ is self-closed, and $\text{prefix}(tr, e)$ includes all events that happen before $e$, therefore, $\text{prefix}(tr, e)$ is self-closed.
- Axiom 2. Assume $f$ is the function for $tr$ that fits Axiom 2, we define $f' = f \setminus \{\text{dom}(\text{prefix}(tr, e))\}$, then $f'$ demonstrates the validate of Axiom 2 for $\text{prefix}(tr, e)$.
- Axiom 3. If this axiom does not holds for $\text{prefix}(tr, e)$, it does not hold for $tr$ either, which contradicts with the assumption.
- Axiom 4. The axiom holds trivial.
- Axiom 5. Assume $e_1$ and $e_2$ are two receive events in $\text{prefix}(tr, e)$, which access the same channels. Since $\text{prefix}(tr, e)$ is self-closed, $e_1, \text{send}$ and $e_2, \text{send}$ are both in the domain of $\text{prefix}(tr, e)$. Since $tr$ is well-formed, $e_1, \text{send} \leq e_2, \text{send}$ holds in the happens before of $tr$.

Since $\text{prefix}(tr, e)$ is a subset of $tr$, therefore the happens before relation in $\text{prefix}(tr, e)$ is a sub-relation of the happens before relation of $tr$. Therefore, $e_1, \text{send} \leq e_2, \text{send}$ also holds in the happens before relation of $\text{prefix}(tr, e)$. \qed
Assume $D = D_1 \parallel D_2$ is a distributed system, which starts from state $(\kappa_1 \cup \kappa_2, \theta)$, $\kappa_1$ is the state of $D_1$ and $\kappa_2$ is for $D_2$. From the view of $D_1$, the state of $D_2$ becomes environment, $\kappa_1$ is the local state, and $\theta$ is the shared state that can be interfered by the environment. Therefore, we project a configuration transition into the view of $D_1$.

$$(D_1, \kappa_1, \theta) \xrightarrow{\lambda(I)} (D'_1, \kappa'_1, \theta')$$

where $\lambda(I) \in \{1, e[I]\}$ represents either a local action (1) or the environmental interference ([e[I]). $I$ is the channel invariant to regulate environmental behaviors.

**Definition 8.** ($D, \kappa, \theta) \xrightarrow{1} (D', \kappa', \theta')$ iff the transition ($D, \kappa, \theta) \rightsquigarrow (D', \kappa', \theta')$ is defined according to the operational semantics given in Fig. 4.

**Definition 9.** $\theta \xrightarrow{0} \theta'$ iff there exists a channel $(i_0, j_0)$ in the domain of $\theta$ and there exists $tr$ that $(-, tr) \models I(i_0, j_0), \theta'(i_0, j_0)$ is obtained by enqueue $tr$ into $I(i_0, j_0)$, and $\theta(i, j) = \theta(i, j)$ holds for all other channels.

**Definition 10.** ($D, \kappa, \theta) \xrightarrow{e[I]} (D', \kappa', \theta')$ iff the following two conditions hold:

- $D = D'$ and $\kappa = \kappa'$; and
- $\theta \xrightarrow{0} \theta'$ and $WF((\kappa', \theta')).$

**Definition 11 (Configuration Safety).** Assume $(\kappa, \theta)$ is well-formed and $\theta = I, (I, 1) \text{SAFE}_0(D, \kappa, \theta, I, q)$ holds always; and (2) $\text{SAFE}_{n+1}(D, \kappa, \theta, I, q)$ holds:

- $\forall i \in \text{dom}(D) \cdot D(i) = \text{skip}$, and let $\kappa' = \kappa \cup \{d(D)\}$, and $\theta' = \theta$; and
- $\neg((D, \kappa, \theta) \xrightarrow{\text{abort}})$; and
- if $(D, \kappa, \theta) \xrightarrow{e[I]} (D', \kappa', \theta')$, then $\theta' = I$ and $\text{SAFE}_n(D', \kappa', \theta')$.

The safety definition follows [20]. Every configuration is safe for zero steps. For $n + 1$ steps, it must satisfies the postcondition if it terminates; not abort; and after any step, maintains the invariant and be safe for another $n$ steps.

**Definition 12 (Semantics).** $I \vdash \{p\} D \{q\}$ iff for any well-formed state $(\kappa, \theta)$, let $\kappa = \kappa \cup \{d(D)\}$, if $\kappa' \models p$ and $\theta = I$, then for any $n \geq 0, \text{SAFE}_n(D, \kappa, \theta, I, q)$.

**Theorem 3 (Soundness).** If $I \vdash \{p\} D \{q\}$, then $I \models \{p\} D \{q\}$.

**Proof:** The soundness is proved by induction over the reasoning rules. By proving the soundness for each rule, the theorem follows by a straightforward rule induction. For brevity, we only show the proofs of the most interesting rules.

**Lemma 4 (Recv Soundness).** Assume well-formed state $(\kappa_0, \theta)$, $\kappa = \kappa_0 \cup \{i\}$, $\kappa_0 \models p$, $\theta = I, I(i, j) = r$, and let $n$ be any natural number; then $\text{SAFE}_n(i \vdash x := \text{recv}(j), \kappa_0, \theta, I, R(j, i, x) \land \circ_{\text{send}} r \land \circ_{\text{pred}} p)$ holds.

**Proof:** We prove by induction:

**Base:** If $n = 0$, $\text{SAFE}_n(i \vdash x := \text{recv}(j), \kappa_0, \theta, I, R(j, i, x) \land \circ_{\text{send}} r \land \circ_{\text{pred}} p)$ holds trivially.

**Inductive step:** Assume the lemma holds for $n = k$, we need to prove $\text{SAFE}_{k+1}(i \vdash x := \text{recv}(j), \kappa_0, \theta, I, R(j, i, x) \land \circ_{\text{send}} r \land \circ_{\text{pred}} p)$.

According to the semantics, $(j, i)$ is in the domain of $I, (D, \kappa_0, \theta)$ does not abort. Assume $(D, \kappa_0, \theta) \xrightarrow{\text{recv}(j)} (D', \kappa'_0, \theta')$, if $\lambda = e[I]$, then $D' = D, \kappa'_0 = \kappa_0$. Moreover, we know that $\theta' = I$ because of Def. 9, so $\text{SAFE}_{k}(i \vdash x := \text{recv}(j), \kappa_0, \theta, I, R(j, i, x) \land \circ_{\text{send}} r \land \circ_{\text{pred}} p)$ holds. On the other hand, if $\lambda = 1$, $\kappa'_0(i) = \text{skip}$, according to the operational semantics and the premise $\theta = I$, and let $\kappa' = \kappa'_0 \setminus \{i\}$, $\kappa' \models R(j, i, x) \land \circ_{\text{send}} r \land \circ_{\text{pred}} p$ also holds.

**Lemma 5 (Send Soundness).** Assume well-formed state $(\kappa_0, \theta), \kappa = \kappa_0 \cup \{i\}, \kappa_0 \models p$, $\theta = I, S(i, j, E) \land \circ_{\text{pred}} p \rightarrow I(i, j)$, and let $n$ be any natural number; then $\text{SAFE}_n(i \vdash \text{send}(E, j), \kappa_0, \theta, I, S(i, j, E) \land \circ_{\text{pred}} p)$ holds.

**Proof:** We prove by induction:

**Base:** If $n = 0$, $\text{SAFE}_n(i \vdash \text{send}(E, j), \kappa_0, \theta, I, S(i, j, E) \land \circ_{\text{pred}} p)$ holds trivially.

**Inductive step:** Assume the lemma holds for $n = k$, we need to prove $\text{SAFE}_{k+1}(i \vdash \text{send}(E, j), \kappa_0, \theta, I, S(i, j, E) \land \circ_{\text{pred}} p)$.

According to the semantics, $(j, i)$ is in the domain of $I, (D, \kappa_0, \theta)$ does not abort. Assume $(D, \kappa_0, \theta) \xrightarrow{\text{send}(E, j)} (D', \kappa'_0, \theta')$, if $\lambda = e[I]$, then $D' = D, \kappa'_0 = \kappa_0$. Moreover, we know that $\theta' = I$ because of Def. 9, so by inductive assumption, $\text{SAFE}_{k}(i \vdash \text{send}(E, j), \kappa'_0, \theta', I, S(i, j, E) \land \circ_{\text{pred}} p)$ holds. On the other hand, if $\lambda = 1$, $\kappa'_0(i) = \text{skip}$, let $\kappa' = \kappa'_0 \setminus \{i\}$, according to the operational semantics, $\kappa' \models S(i, j, E) \land \circ_{\text{pred}} p$, and a new trace added to the end of $\theta(i, j)$, by the premise $S(i, j, E) \land \circ_{\text{pred}} p \rightarrow I(i, j)$, $\theta' = I$, so $(\kappa', \theta')$ is well-formed.

**Lemma 6 (PAR Soundness).** If $\text{SAFE}_n(D_1, \kappa, \theta, I, q_1), \text{SAFE}_n(D_2, \kappa, \theta, I, q_2), \text{and freeVar}(p_1) \cap \text{freeVar}(p_2) = \emptyset$, then $\text{SAFE}_n(D_1 \parallel D_2, \kappa, \theta, q_1 \parallel q_2)$.

**Proof:** By induction on $n$. In the inductive step, we know

$$IH(n) \overset{\text{def}}{=} \forall D_1, \kappa, \theta, q_1, q_2 \cdot WF(\kappa, \theta) \land \text{SAFE}_n(D_1, \kappa, \theta, I, q_1) \land \text{SAFE}_n(D_2, \kappa, \theta, I, q_2) \land \text{freeVar}(p_1) \cap \text{freeVar}(p_2) = \emptyset \Rightarrow \text{SAFE}_n(D_1 \parallel D_2, \kappa, \theta, q_1 \parallel q_2).$$

holds and we have to show $IH(n + 1)$. Pick arbitrary program $D_1, D_2$, and global state $\kappa$ and assume

1. $\text{SAFE}_{n+1}(D_1, \kappa, \theta, I, q_1)$,
2. $\text{SAFE}_{n+1}(D_2, \kappa, \theta, I, q_2)$,
3. the variable side conditions, and try to show

**1)** If all threads in $D_1$ and $D_2$ are skip, then trivial.

**2)** If $(D_1 \parallel D_2, \kappa, \theta) \rightsquigarrow \text{abort}$, according to the operational semantics, $(D_1, \kappa, \theta) \rightsquigarrow \text{abort}$ or $(D_2, \kappa, \theta) \rightsquigarrow \text{abort}$, this contradicts with our assumption (1) and (2).

**3)** Assume $(D_1 \parallel D_2, \kappa, \theta) \xrightarrow{e[I]} (D', \kappa', \theta')$, then the operational semantics has three possible transitions for $D_1 \parallel D_2$. 
The first case is if $\lambda = e$, $D' = D_1 \| D_2$ and $\kappa' = \kappa$, according to Def. 9, $\theta' \models I$, $\text{SAFE}_n(D', \kappa, \theta', q_1 \oplus q_2)$ holds by induction. Therefore, according to the safety definition, $\text{SAFE}_{n+1}(D_1 \| D_2, \kappa, I, q_1 \oplus q_2)$ holds.

The second case is $\lambda = 1(I)$, $D' = D'_1 \| D_2$ and $(D_1, \kappa, \theta) \xrightarrow{\lambda} (D'_1, \kappa', \theta')$. From (1), there exists $\kappa'$ and $\theta'$ that $\text{SAFE}_n(D'_1, \kappa', \theta', I, q_1)$. From (2), $D_2$ is safe within $n + 1$ steps, we have $\text{SAFE}_{n+1}(D_2, \kappa', I, q_2)$, and $\text{SAFE}_n(D_2, \kappa', I, q_2)$ holds trivially. By inductive assumption, $\text{SAFE}_n(D'_1 \| D_2, \kappa', I, q_1 \oplus q_2)$.

The third case is $\lambda = 1(I)$, and $D' = D_1 \| D'_2$, which is completely symmetric with the second case. \qed