Scientific Annals of Computer Science vol. 24 (2), 2014, pp. 217–252
doi: 10.7561/SACS.2014.2.217

Rely-Guarantee Based Reasoning for Message-Passing Programs[1]

Jinjiang Lei[2], Zongyan Qiu[2]

Abstract

The difficulties of verifying concurrent programs lie in their inherent non-determinism and interferences. Rely-Guarantee reasoning is one useful approach to solve this problem for its capability in formally specifying inter-thread interferences. However, modern verification requires better locality and modularity. It is still a great challenge to verify a message-passing program in a modular and composable way.

In this paper, we propose a new reasoning system for message-passing programs. It is a novel logic that supports Hoare style triples to specify and verify distributed programs modularly. We concretize the concept of event traces to represent interactions among distributed agents, and specify behaviors of agents by their local traces with regard to environmental assumptions — an idea inspired by Rely-Guarantee reasoning. Based on trace semantics, the verification is compositional in both temporal and spatial dimensions. To show validity, we apply our logic to modularly prove several examples.

Keywords: trace semantics, message passing, rely guarantee

[1]The work is supported by NNSF of China, Grant No. 61272160, 61100061, and 61202069. A preliminary version of this paper is published in the Proceedings of The 11th International Colloquium on Theoretical Aspects of Computing, LNCS 8687, 277-294, Springer, 2014.

[2]LMAM and Department of Informatics, School of Math, Peking University, 5 Yiheyuan Road, Haidian District, Beijing, China, E-mail: {jinjiang.lei, zyqiu}@pku.edu.cn

1 Introduction

With the blossom of multi-core processors and large scale network communication, concurrency has become a crucial element in software systems. According to the ways of inter-thread communication, in general, there are two concurrent models: shared memory and message-passing. Applications using either of the two models are notoriously difficult to be verified because of non-deterministic interleaves of memory accesses or message passing.

1.1 Separation Logic and Rely-Guarantee Reasoning

Separation Logic (SL) is introduced by Reynolds *et al.* (e.g., [19]) for specifying and verifying programs which manipulate states with complex structures, especially the pointers and data structures. The key idea of SL is to include in the logic language some connectors to describe explicitly the separation of parts in the program states. By migrating the ideas of SL, verification of shared memory models has gained great progress. Some new logics have been developed and their power has been well respected, e.g., Concurrent Separation Logic (CSL) [2], concurrent abstract predicate [5], and lots of other separation-based reasoning [20, 21].

Separation Logic supports local reasoning for separated parts of programs states. For the concurrent programs, when different threads reside in separated regions, their state separation enables local reasoning naturally. However, despite of state separation, the core feature of concurrency is the interferences among separated threads. Thus, for the formal reasoning, the crucial difficulty is how to formally specify and verify the interferences.

One pioneer work in this area is Rely-Guarantee (RG) reasoning [8]. In RG reasoning, the specification of a program D takes the form of:

$$\mathcal{R}, \mathcal{G} \vdash \{p\}\, D\, \{q\}$$

where $\mathcal{R}$ and $\mathcal{G}$ are the rely and guarantee conditions respectively; p and q are pre- and post-assertions for D's local state. In the specification, $\mathcal{R}$ specifies the set of state transitions that the *environment* (the other threads in the program) could apply to the local state of D. Since environmental behavior is non-deterministic, p and q are required to be *stable* under the interference of $\mathcal{R}$, that is, no matter how local states transit caused by the environment, the validity of p and q is fixed. On the other hand, $\mathcal{G}$ represents the set of all possible state transition that are caused by the local execution of D.

One group of distinguished idea in this area is the combination of Separation Logic and Rely-Guarantee reasoning, e.g., SAGL [6] and RGSep [20]. For instance, the parallel composition rule in RGSep takes the following form:

$$\frac{[\![G_2]\!] \subseteq [\![\mathcal{R}_1]\!] \quad \mathcal{R}_1, \mathcal{G}_1 \vdash \{p_1\}\, D_1\, \{q_1\} \qquad [\![G_1]\!] \subseteq [\![\mathcal{R}_2]\!] \quad \mathcal{R}_2, \mathcal{G}_2 \vdash \{p_2\}\, D_2\, \{q_2\}}{\mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2 \vdash \{p_1 * p_2\}\, D_1 || D_2\, \{q_1 * q_2\}}$$

where $[\![\bullet]\!]$ denotes the semantics of $\bullet$, and $*$ is the separating conjunction operator that specifies *separated* states [19]. This rule says that two programs are compatible with each other when they mutually include the other one's guarantee condition in their rely condition. The rely condition of the combined program is the intersection of $\mathcal{R}_1$ and $\mathcal{R}_2$, and the guarantee condition is the union of $\mathcal{G}_1$ and $\mathcal{G}_2$.

As presented, existing methods that applies RG reasoning to shared memory still encounter several problems that greatly impede its application:

- Pre- and post-assertions are confined by the "stable" requirement, which is quite strict that forbids users from writing assertions freely.
- The rely and guarantee conditions are globally predefined. This require users to carefully consider all system behavior details beforehand, which cannot be changed during reasoning.

In the rest of the paper, we apply RG reasoning to message-passing model, which addresses the above problems.

1.2 Lamport's Graph Model

Like SAGL and RGSep, many state-based verifications of shared memory concurrency have been studied. But for message-passing models, although it have been extensively studied using various process calculi [7, 14, 16], fewer Hoare-style state-based reasoning systems are developed, especially systems supporting modular reasoning. These are the main focus of our work.

To develop such a verifying system, the first step is to define a model for the program states. In this paper, we choose the classic Lamport's event graph [10]. Lamport introduced *event graphs* (or *event traces*) as a representation for the semantics of message-passing programs. Event graphs are essentially Directed Acyclic Graphs (DAGs) composed by nodes and directed arrows, where nodes represent atomic actions (e.g., send/receive events), and directed arrows represent inter-agent communications. Each

event graph defines implicitly a partial order — *happens-before*, denoted by $\prec$ — among nodes, which is the transitive closure combined of agents' local order and directed arrows, to reveal the *causality relation* among events. This order will be formally defined later.

1.3 Our Approach

In terms of formal verification, for any semantics based on event graphs, one crux is to modularly specify graph structures. As a graph represents a collection of ordered and correlated events (nodes), the modularity could only be achieved when we can carve out irrelevant events in reasoning a local behavior. To make this possible, we view the structure of an event graph from two dimensions: the spatial dimension and the temporal dimension. We introduce a *separating conjunction* operator, $*$, to depict the spatial dimension by specifying separated traces; and an operator, $\circ$, to represent *sequential conjunction*, which defines the temporal dimension based on happens-before relation $\prec$, and takes a stronger condition than the spatial one.

Our logic adopts the Hoare style triples to specify message-passing programs, and makes local reasoning of the programs a reality. Generally, the semantics of a set of agents D is specified by a triple as follows:

$$\{r, p\}\, D\, \{r', q\}$$

where r and r' specify D's expectations (or assumptions) about its *environment* (i.e., the behaviors of other agents, similar to the rely condition in RG reasoning), and p and q specify the local states (changes) of D. The reasoning of D relies on its environmental assumptions, and the local behaviors of D are required to be correlated with its environmental expectations. Local agents are able to dynamically calculate and strengthen its environmental assumption in r' in order to fulfill certain local function that are specified. On the other hand, the other agents should satisfy the expectation of D in order to parallel composite with D.

We present the proof of a tiny example to show the deduction of our system and how the spatial and temporal modularity is achieved. We take the following simple program as the example:

```
send (2, pt);          ||   x := recv (pt);
send (3, pt);          ||   y := recv (pt);
```

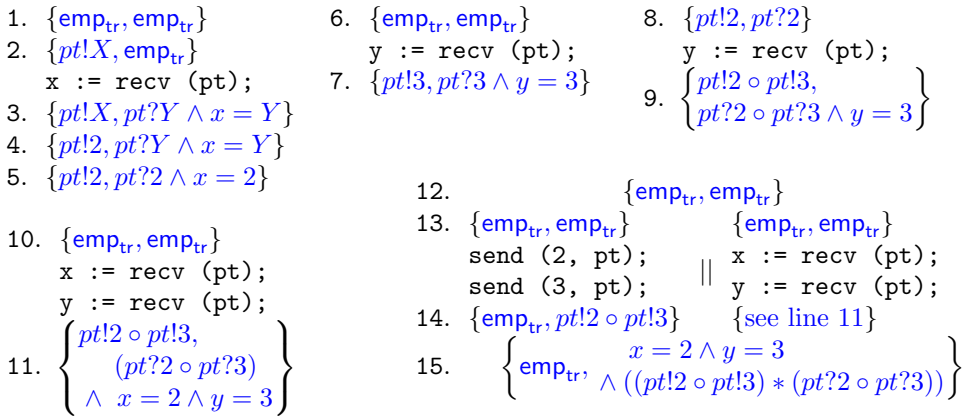


Figure 1: Modular Proof of a Tiny Example

The left agent sends messages 2 and 3 to the port pt sequentially, and the right agent is the owner of pt, who withdraws the messages and stores them into its local variable `x` and `y` respectively.

The proof given in Fig. 1 is modular, because the system is proved agent by agent, and an agent is separately proved command by command.

Lines 1 – 5 are the proof for the first receive command: It starts from $\{\mathsf{emp}_{\mathsf{tr}}, \mathsf{emp}_{\mathsf{tr}}\}$, where no assumption for the environment is made (the first $\mathsf{emp}_{\mathsf{tr}}$) and no local action is taken (the second $\mathsf{emp}_{\mathsf{tr}}$). Then in line 2, we assume the environment sends message X to pt ($pt!X$), where X is an implicitly existential-quantified logical variable and its scope is confined within the environmental part. After executing the receive command, there is a receive event in the local state ($pt?Y \wedge x = Y$), where Y is existential-quantified to represent the message received and its scope is the local state. Line 4 is deduced from line 3 by strengthening the environmental assumption ($pt!2 \Rightarrow \exists X \cdot pt!X$); and line 5 comes because a receive should match with its sender ($pt!2 * pt?Y \Rightarrow Y = 2$)[3].

The other receive is proved separately (lines 6 – 7) which is similar as lines 1 – 5. Lines 8 – 9 are obtained from lines 6 – 7 by adding a "frame" — $pt!2 * pt?2$ — ahead of current event graph, where $pt!2$ is added ahead of the environment, and $pt?2$ is ahead of the local state. Note that the frame does not affect existing proofs and program state. The "ahead of" relation is formally called "happens-before", which is served by the operator "$\circ$". We

[3]This form of writing is just for easy understanding, precisely it should be $pt?Y \overset{pt!2}{\Longrightarrow} pt?2$, which will be formally discussed in Section 5.

can add another frame "$x = 2$" to the local state in line 8 and 9 in order to ensure line 8 is the same as line 5. This step is trivial and thus omitted.

Having the proofs for the two receives, the whole specification of the first agent is obtained by sequentially combining the two proofs (lines 10 – 11).

The specification of the sending agent shown in lines 13 – 14 is trivial and needs no explanation. The composition for the two agents is shown in lines 12 – 15. In line 15, the environmental assumption of the receiver (line 11) is satisfied by the local state of the sender (line 14), therefore, the environmental part of line 15 is $\mathsf{emp}_{\mathsf{tr}}$. The local part of line 15 is just the composition of local state of both agents.

1.4 Contributions

In summary, the logic we developed in this work makes the following contributions:

- It concretizes the concept of *event graph* [10] to represent the interactions among agents, and proposes a set of trace predicates to specify the properties of traces.
- It supports two-dimensional modularity: temporal modularity, as shown by the separated proofs of the receives in above example; and spatial modularity, as shown by the separated proofs of the agents.
- Each agent can be proved locally with suitable explicitly calculated assumptions about its environment, and proofs of separate agents can be combined as long as their local behaviors could mutually satisfy the environmental assumptions of their partners.
- Comparing with classic RG reasoning, our method removes the stability requirement of pre/post-assertions, and allows dynamically calculate environment assumption. These relaxation strongly ease the burden of users.

This article extends the conference paper which is presented in ICTAC 2014 [11]. There are two major improvements. First, we significantly expand the details for reasoning distributed programs that exhibits non-deterministic behaviors. Second, we prove another message-passing algorithm – leader election – to further validate our theory.

In the rest of the paper, we give a formal definition of event trace, and present a trace algebra for separating and sequential conjunctions in Section 2; and a formal presentation for the model and operational semantics in Section 3. The assertion language for specifying trace structures and the reasoning logic was presented at Sections 4 and 5. Formal semantics are presented and proved in Section 6. Section 7 specifically shows the application for proving non-deterministic programs. Sections 8 and 9 give case studies, and Section 10 discusses the related work and concludes.

2 Event Trace – the Basic Program State Settings

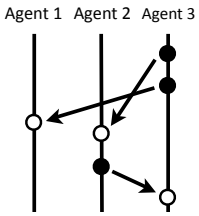
A distributed system is composed by a set of *agents*[4], which are run concurrently. Each of the agents represents a computational process that can own several *ports* for receiving messages. In our setting, each port belongs to one agent, while an agent can own multiple ports. We adopt asynchronous message passing: send commands will not be blocked, while receives will be blocked if there is no message received on the designated ports. We assume the state of a port is a queue, and messages are transmitted following the FIFO-principle [3].

2.1 Traces

We use *event graphs* [10] to depict the semantics of distributed programs. The left part of Fig. 2 shows the trace of a program execution, where solid nodes represent send events and hollow nodes are receives. The picture also reveals that event traces are time-space graphs, where space is measured by agents, and time is measured by agent local orders and inter-agent arrows (which represent inter-agent communications).

The state of traces is defined formally in Fig. 2 (right part). Each event is represented by a tuple with four components, and referred by a unique reference e. A trace tr is a map from event references to its corresponding event, $tr(e) = (m, pt, pd, sd)$. In an event, component m (referred by $e.\mathsf{val}$) and pt (referred by $e.\mathsf{port}$) are the value and port for the message respectively; pd (referred by $e.\mathsf{pred}$) is e's local direct predecessor; and sd (referred by $e.\mathsf{src}$) refers to e's corresponding send event when e is a receive. If e is the

[4]The *agent* might be called *process*, or *thread*, etc. in other works. We will use the name *agent* consistently in this work.



$$\text{Value} = \text{Int} \qquad \text{AgntID} = \text{Nat} \qquad \text{Port, EvntID : Discrete Types}$$

$$\text{(Event)} \quad (m, pt, pd, sd) : \quad \text{Val} \times \text{Port} \times \text{EvntID} \cup \{\text{AgntID}\} \times \text{EvntID} \cup \{\mathsf{nil}\}$$

$$\text{(EvntTrc)} \qquad tr : \text{EvntID} \rightharpoonup_{\text{fin}} \text{Event}$$

Figure 2: Trace State

first event of an agent, $e.\mathsf{pred}$ is the agent ID where e lives in; and $e.\mathsf{src} = \mathsf{nil}$ if e is a send event.

We use $\mathsf{isSend}(e)$ and $\mathsf{isRecv}(e)$ to specify the type of e:

$$\mathsf{isSend}(e) \stackrel{\text{def}}{=} e.\mathsf{src} = \mathsf{nil} \qquad \mathsf{isRecv}(e) \stackrel{\text{def}}{=} e.\mathsf{src} \neq \mathsf{nil}$$

We recursively define a function $\mathsf{agent}(e)$ to return the agent ID of the event referred by e:

$$\mathsf{agent}(e) \stackrel{\text{def}}{=} \begin{cases} e.\mathsf{pred} & e.\mathsf{pred} \in \text{AgntID} \\ \mathsf{agent}(e.\mathsf{pred}) & \text{otherwise} \end{cases}$$

2.2 Well-formed Traces

Event traces are specific structures to record the communicating history among agents. In this section we present an axiomatic definition of traces. As Lamport postulated [10], events are partially ordered by *happens-before* relation $\prec$.

Definition 1 (Happens-Before) *The* happens-before *relation of* tr, $\prec$, *is:*

$$\begin{aligned} e \prec_{\text{step}} e' &\stackrel{\text{def}}{=} e = e'.\mathsf{pred} \vee e = e'.\mathsf{src} \\ e \prec e' &\stackrel{\text{def}}{=} \exists e'' \cdot e \prec e'' \wedge e'' \prec_{\text{step}} e') \\ &\textit{where } \{e, e'\} \subseteq \mathsf{dom}(tr) \end{aligned}$$

Based on Def. 1, we define six axioms to specify *well-formed traces*. Axioms 1 – 4 are general axioms which are proposed originally in [1]; axioms 5 – 6 are specifically proposed for our model. We use tr to denote an event trace.

Axiom 1 *tr is self-closed:*

$$\forall e \in \mathsf{dom}(tr) \cdot e.\mathsf{pred} \notin \mathrm{AgntID} \Rightarrow e.\mathsf{pred} \in \mathsf{dom}(tr), \text{ and}$$
$$\forall e \in \mathsf{dom}(tr) \cdot e.\mathsf{src} \neq \mathsf{nil} \Rightarrow e.\mathsf{src} \in \mathsf{dom}(tr).$$

Axiom 2 *Happens-before relation $\prec$ is strongly well founded. There exists a function $f : \mathsf{dom}(tr) \to \mathrm{Nat}$ such that:*

$$\forall e, e' \in \mathsf{dom}(tr) \cdot e \prec e' \Rightarrow f(e) < f(e').$$

Axiom 3 *Maps $\bullet.\mathsf{pred}$ and $\bullet.\mathsf{src}$ are injective:*

$$\forall e, e' \in \mathsf{dom}(tr) \cdot e.\mathsf{pred} = e'.\mathsf{pred} \Rightarrow e = e', \text{ and}$$
$$\forall e, e' \in \mathsf{dom}(tr) \cdot e.\mathsf{src} = e'.\mathsf{src} \wedge e.\mathsf{src} \neq \mathsf{nil} \Rightarrow e = e'.$$

Axiom 4 *The* send *field of a receive event refers to its corresponding send event:*

$$\forall e \in \mathsf{dom}(tr) \cdot \mathsf{isRecv}(e) \Rightarrow \exists e' \cdot \begin{array}{c} e.\mathsf{src} = e' \wedge \mathsf{isSend}(e') \\ \wedge\, e.\mathsf{val} = e'.\mathsf{val} \wedge e.\mathsf{port} = e'.\mathsf{port}. \end{array}$$

Axiom 5 *Communications are robust that there is no lost message. Let e_1 and e_2 be two send events:*

$$e_1 \prec e_2 \wedge e_1.\mathsf{port} = e_2.\mathsf{port} \ \wedge \exists e_2' \cdot e_2'.\mathsf{src} = e_2 \Rightarrow \exists e_1' \cdot e_1'.\mathsf{src} = e_1.$$

That is, if e_2 is received, all send events that happen before e_2 on the same channel must have been received.

Axiom 6 *Messages are sent and received by the FIFO principle. Let e_1 and e_2 be two send events:*

$$e_1 \prec e_2 \wedge e_1.\mathsf{port} = e_2.\mathsf{port} \wedge e_1'.\mathsf{src} = e_1 \wedge e_2'.\mathsf{src} = e_2 \Rightarrow \neg(e_2' \prec e_1').$$

We use $\mathcal{A}_1, \ldots, \mathcal{A}_6$ to represent the above axioms, and $\mathcal{A}$ to denote their conjunction: $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_6$.

Theorem 1 *Let Prop be the type of propositions over tr, then for all P : $\mathrm{EvntID} \to \mathrm{Prop}$:*

$$(\forall e' \cdot (\forall e \cdot e \prec e' \wedge P(e) \to P(e'))) \Rightarrow \forall e \cdot P(e)$$

Theorem 1 is the induction over event traces: take any event e' in the trace, if all events that happen before e' satisfy P leads the truth of P at e', then P holds all over the trace. This theorem is useful in proving complex properties over traces and also application in our method; [1] gives many examples showing its usage.

Definition 2 (Well-formed Trace) *Trace tr is well-formed, $WF(tr)$, iff there exist tr' and tr'' such that:*

$$tr'' = tr \uplus tr'^{5} \wedge tr'' \models \mathcal{A}$$

That is, any well-formed trace, tr, is a sub-trace of some "complete" trace tr'' such that tr'' entails $\mathcal{A}$.

In our model, events stand for the atomic actions for sending/receiving messages. Local actions, e.g., variable assignment, control flow execution *etc.*, are not recorded as events in the graph. Axioms and theorems in this section are crucial for trace implication, e.g., $p \Rightarrow q$.

2.3 Trace Separation and Algebra

To structurally specify event traces, we introduce two operators, separating conjunction $*$ and sequential conjunction $\circ$, where:

$tr_1 * tr_2$ is the union of all the events in tr_1 and tr_2 as long as tr_1 and tr_2 contain disjointed set of events:

$$tr * tr' \stackrel{\text{def}}{=} tr \uplus tr' \quad \text{iff} \quad WF(tr \uplus tr')$$

$tr_1 \circ tr_2$ returns $tr_1 * tr_2$ if three additional conditions hold: (1) no event in tr_2 happens before any event in tr_1; (2) if e_1 ($\in tr_1$) and e_2 ($\in tr_2$) send messages to the same port, then e_1 happens before e_2; and (3) if e_1 ($\in tr_1$) and e_2 ($\in tr_2$) receive messages from the same port, then e_1 happens before e_2.

$$\begin{aligned} tr \circ tr' \stackrel{\text{def}}{=} tr * tr' \quad \text{iff} \quad & \forall e \in \mathsf{dom}(tr), e' \in \mathsf{dom}(tr') \cdot \\ & \neg(e' \prec e) \wedge (\mathsf{isSend}(e) \wedge \mathsf{isSend}(e') \wedge e.\mathsf{port} = e'.\mathsf{port} \Rightarrow e \prec e') \\ & \wedge (\mathsf{isRecv}(e) \wedge \mathsf{isRecv}(e') \wedge e.\mathsf{port} = e'.\mathsf{port} \Rightarrow e \prec e') \end{aligned}$$

[5] $f \uplus g$ is the union of f and g but requires that f and g have disjointed domains.

$$tr_1 = tr_2 \Rightarrow tr_1 * tr_3 = tr_2 * tr_3 \qquad tr_1 * tr_2 = tr_2 * tr_1$$
$$tr_1 * tr_2 = tr_1 * tr_3 \Rightarrow tr_2 = tr_3 \qquad tr = tr_1 \circ tr_2 \Rightarrow tr = tr_1 * tr_2$$
$$tr_1 \circ (tr_2 \circ tr_3) = (tr_1 \circ tr_2) \circ tr_3 \qquad tr_1 * (tr_2 * tr_3) = (tr_1 * tr_2) * tr_3$$
$$tr = tr_1 \circ (tr_2 * tr_3) \Rightarrow tr = (tr_1 \circ tr_2) * tr_3$$
$$tr = (tr_1 * tr_2) \circ tr_3 \Rightarrow tr = (tr_1 \circ tr_3) * tr_2$$

Figure 3: Selected Properties for Traces

In [22], Wehrman *et al.* defined another semantics for $tr \circ tr'$, which only requires events in tr' do not happen before events in tr. Our semantic definition of *sequential composition* is stronger. Take the trace $(pt!88 \circ pt!14) * (pt?x \circ pt?y)$ for instance, we can deduce $x = 88 \wedge y = 14$ with the additional conditions, otherwise the $x = 14 \wedge y = 88$ is permitted as well.

The two operators satisfy certain commutative and associative laws. Fig. 3 lists some selected properties for trace structures, which are sound based on the semantics.

3 Programming Language

In this section, we define the programming language used for constructing the distributed programs (system models) and its operational semantics.

Fig. 4 gives the syntax of the language. We use E and B to denote numerical and boolean expressions. Command $\mathbf{send}\,(E, pt)$ sends message E to port pt; and $x := \mathbf{recv}\,(pt)$ withdraws a message from pt and stores it into the local variable x. A distributed program is a parallel composition of agents C_i, where each agent is tagged with a unique agent ID ($i_1, \ldots, i_k$ in Fig. 4). For simplicity, we don't consider memory management in our model.

The program state is defined in Fig. 5. We choose the simplest setting that defines the state composed by only a store s and a trace tr, where s maps variable names to values, and tr is already defined in Fig. 2.

The operational semantics is defined by a set of rules which describe configuration transitions caused by the program execution. These rules take the following form:

$$(D, s, tr) \rightsquigarrow (D', s', tr')$$

$$
\begin{array}{llll}
\text{(Expr)} & E & ::= & x \mid X \mid n \mid E + E \mid E - E \mid \ldots \\
\text{(BExp)} & B & ::= & \textbf{true} \mid \textbf{false} \mid E = E \mid E \neq E \mid \ldots \\
\text{(Comd)} & c & ::= & x := E \mid \textbf{skip} \mid \textbf{send}\,(E, pt) \mid x := \textbf{recv}\,(pt) \\
\text{(Stmts)} & C & ::= & c \mid C_1; C_2 \mid \textbf{while}\ B\ \textbf{do}\ C \mid \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2 \\
\text{(Prog)} & D & ::= & i_1 : C_1 \,||\, \ldots \,||\, i_k : C_k
\end{array}
$$

Figure 4: The Language

$$
\begin{array}{llll}
\multicolumn{4}{l}{\text{Loc} = \text{Int} \qquad \text{Var} : \text{Discrete Type}} \\
(\mathit{Store}) & s & : & \text{Var} \rightharpoonup_{\text{fin}} \text{Value} \\
(\mathit{EvntTrc}) & tr & : & \text{EvntID} \rightharpoonup_{\text{fin}} \text{Event} \\
(\mathit{State}) & \sigma & ::= & (s, tr)
\end{array}
$$

Figure 5: State Definition

If there is only one agent i ($D = i : C$), the transition can take the form of:

$$(C, s, tr) \rightsquigarrow_i (C', s', tr')$$

Fig. 6 gives the operational semantics. In the rules, $\{i_1 : v_1; \ldots; i_n : v_n\}$ denotes a function f with $\mathsf{dom}(f) = \{i_1, \ldots, i_n\}$ and $f(i_j) = v_j$; $f\{x \hookrightarrow v\}$ remaps x of f to v; $f \uplus g$ is function union when $\mathsf{dom}(f) \cap \mathsf{dom}(g) = \varnothing$; $[\![E]\!]_s$ and $[\![B]\!]_s$ evaluate the numerical and boolean expressions based on store s.

In order to construct valid traces, each new event should be linked to its predecessor – the last event of the current trace. Function $\mathsf{last}(tr, i)$ returns the last event of agent i in tr. If there is no event at agent i, then the function returns "i" since the $\bullet.\mathsf{pred}$ field of the first event is an agent ID. Since all events on a same agent are totally ordered by the happens-before relation "$\prec$", $\mathsf{last}(tr, i)$ always return a unique element.

$$
\mathsf{last}(tr, i) \stackrel{\text{def}}{=} \begin{cases} i & \{e \in \mathsf{dom}(tr) \mid \mathsf{agent}(e) = i\} = \varnothing \\ \max\limits_{\prec} \left\{ e \;\middle|\; \begin{array}{r} e \in \mathsf{dom}(tr) \\ \wedge\, \mathsf{agent}(e) = i \end{array} \right\} & \text{otherwise} \end{cases}
$$

For receive events, since the model adopts FIFO message passing rules, a new receive should match with the oldest message remaining in the designated port. We define a predicate $\mathsf{fstUnMchd}(e, tr, pt)$ to state that e is the first pending send event on port pt, that is, e is a send event on pt which has not matched with a receive yet, and no other unmatched send event on pt

$$\frac{[\![E]\!]_s = n}{(x := E, s, tr) \rightsquigarrow_i (\mathbf{skip}, s\{x \hookrightarrow n\}, tr)} \quad \frac{[\![E]\!]_s \text{ undefined}}{(x := E, s, tr) \rightsquigarrow_i \mathbf{abort}}$$

$$\frac{\mathsf{fstUnMchd}(e, tr, pt) \quad e.\mathsf{val} = n \quad e' = \mathsf{last}(tr, i) \quad e'' \notin \mathsf{dom}(tr)}{(x := \mathbf{recv}\,(pt), s, tr) \rightsquigarrow_i (\mathbf{skip}, s\{x \hookrightarrow n\}, tr \uplus \{e'' : (n, pt, e', e)\})}$$

$$\frac{[\![E]\!]_s = n \quad e = \mathsf{last}(tr, i) \quad e' \notin \mathsf{dom}(tr)}{(\mathbf{send}\,(E, pt), s, tr) \rightsquigarrow_i (\mathbf{skip}, s, tr \uplus \{e' : (n, pt, e, \mathsf{nil})\})}$$

$$\frac{[\![E]\!]_s \text{ undefined}}{(\mathbf{send}\,(E, pt), s, tr) \rightsquigarrow_i \mathbf{abort}} \quad \frac{[\![B]\!]_s = \mathbf{true}}{(\mathbf{if}\,B\,\mathbf{then}\,C_1\,\mathbf{else}\,C_2, s, tr) \rightsquigarrow_i (C_1, s, tr)}$$

$$\frac{[\![B]\!]_s = \mathbf{false}}{(\mathbf{if}\,B\,\mathbf{then}\,C_1\,\mathbf{else}\,C_2, s, tr) \rightsquigarrow_i (C_2, s, tr)}$$

$$\frac{[\![B]\!]_s = \mathbf{true}}{(\mathbf{while}\,B\,\mathbf{do}\,C, s, tr) \rightsquigarrow_i (C; \mathbf{while}\,B\,\mathbf{do}\,C, s, tr)}$$

$$\frac{[\![B]\!]_s = \mathbf{false}}{(\mathbf{while}\ B\ \mathbf{do}\ C, s, tr) \rightsquigarrow_i (\mathbf{skip}, s, tr)} \quad \frac{(D_1, \sigma) \rightsquigarrow \mathbf{abort} \ \text{ or } \ (D_2, \sigma) \rightsquigarrow \mathbf{abort}}{(D_1 \,||\, D_2, \sigma) \rightsquigarrow \mathbf{abort}}$$

$$\frac{(D_1, \sigma) \rightsquigarrow (D_1', \sigma')}{(D_1 \,||\, D_2, \sigma) \rightsquigarrow (D_1' || D_2, \sigma')} \quad \frac{(D_2, \sigma) \rightsquigarrow (D_2', \sigma')}{(D_1 \,||\, D_2, \sigma) \rightsquigarrow (D_1 || D_2', \sigma')}$$

$$\frac{(C_1, \sigma) \rightsquigarrow_i (C_1', \sigma')}{(C_1; C_2, \sigma) \rightsquigarrow_i (C_1'; C_2, \sigma')} \quad \frac{(C_1, \sigma) \rightsquigarrow_i \mathbf{abort}}{(C_1; C_2, \sigma') \rightsquigarrow_i \mathbf{abort}} \quad \frac{}{(\mathbf{skip}; C, \sigma) \rightsquigarrow_i (C, \sigma)}$$

Figure 6: Operational Semantics

happens before e. Formally:

$$\begin{aligned} \mathsf{rcvd}(e, pt) &\stackrel{\text{def}}{=} \mathsf{isSend}(e) \wedge \exists e' \cdot e'.\mathsf{src} = e \\ \mathsf{fstUnMchd}(e, tr, pt) &\stackrel{\text{def}}{=} \neg\mathsf{rcvd}(e, pt) \wedge \neg\exists e' \cdot (\neg\mathsf{rcvd}(e', pt) \wedge e' \prec e) \end{aligned}$$

Semantics of local primitives, e.g., assignment, control flow commands are regular. Rules for send and receive primitives are added, which create new events in the trace. For a receive command, it should find the first unmatched send event on the port in tr according to $\mathsf{fstUnMchd}(e, tr, pt)$, then create the corresponding receive event and add it to the trace.

For the semantics, we have the following result:

Theorem 2 *Let (D, σ) be the initial state, and σ_{tr} is the trace component of σ, then the execution traces of any distributed program would entail $\mathcal{A}$:*

$$((D, \sigma) \rightsquigarrow^* (D', \sigma')) \wedge \sigma_{tr} \models \mathcal{A} \Rightarrow \sigma'_{tr} \models \mathcal{A}.$$

$$(s,tr) \models \mathsf{emp_{tr}} \text{ iff } \mathsf{dom}(tr) = \varnothing \qquad (s,tr) \models \mathsf{true_{tr}} \text{ iff } WF(tr)$$
$$(s,tr) \models B \text{ iff } [\![B]\!]_s = \mathbf{true}$$
$$(s,tr) \models pt!E \text{ iff } \exists n, e \cdot [\![E]\!]_s = n \wedge \mathsf{dom}(tr) = \{e\} \wedge tr(e) = (n, pt, \_, \mathsf{nil})$$
$$(s,tr) \models pt?E \text{ iff } \exists n, e \cdot [\![E]\!]_s = n \wedge \mathsf{dom}(tr) = \{e\} \wedge tr(e) = (n, pt, \_, \neg\mathsf{nil})$$

$$(s,tr) \uplus (s',tr') \stackrel{\text{def}}{=} \begin{cases} (s \uplus s', tr * tr') & \text{if } s \uplus s' \wedge tr * tr' \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(s,tr) \odot (s',tr') \stackrel{\text{def}}{=} \begin{cases} (s, tr \circ tr') & \text{if } s = s' \wedge tr \circ tr' \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$tr^* \stackrel{\text{def}}{=} \varnothing \cup tr \cup tr \circ tr \cup \ldots$$
$$\sigma^* \stackrel{\text{def}}{=} (s, tr^*) \quad \text{where } \sigma = (s, tr)$$

$$\sigma \models p_1 * p_2 \text{ iff } \exists \sigma_1, \sigma_2 \cdot \sigma_1 \uplus \sigma_2 = \sigma \wedge \sigma_1 \models p_1 \wedge \sigma_2 \models p_2$$
$$\sigma \models p_1 \circ p_2 \text{ iff } \exists \sigma_1, \sigma_2 \cdot \sigma_1 \odot \sigma_2 = \sigma \wedge \sigma_1 \models p_1 \wedge \sigma_2 \models p_2$$
$$\sigma \models p^* \text{ iff } \exists \sigma' \cdot \sigma' \models p \wedge \sigma = \sigma'^* \qquad \sigma \models \neg p \text{ iff } \sigma \not\models p$$
$$\sigma \models p \wedge q \text{ iff } \sigma \models p \wedge \sigma \models q \qquad \sigma \models p \Rightarrow q \text{ iff if } \sigma \models p, \text{then } \sigma \models q$$
$$\sigma \models \exists X \cdot p \text{ iff } \exists n \in \mathrm{Val} \cdot \sigma \models p[n/X]$$

Figure 7: Semantics of Assertions

4 Assertion Language

This section defines the assertion language for event traces. We assume an infinite set of logical variables $LVar = \{X, Y, \ldots\}$. The assertion language is a mixture of store predicates, and trace predicates with the following syntax:

$$\begin{array}{rll} p, q ::= & E = E \mid E > E \mid \ldots & \text{(store predicates)} \\ & \mid \mathsf{emp_{tr}} \mid \mathsf{true_{tr}} \mid pt!E \mid pt?E & \text{(trace predicates)} \\ & \mid \neg p \mid p \wedge q \mid \exists X \cdot p \mid p * q \mid p \circ q \mid p^* \mid \ldots & \text{(connectives)} \end{array}$$

The semantics of assertions is defined in Fig. 7, where $\mathsf{emp_{tr}}$ and $\mathsf{true_{tr}}$ specify empty trace and any well-formed trace respectively. For the primitive assertions, $pt!E$ and $pt?E$ specify singleton events, where $pt!E$ represents sending message E to pt and $pt?E$ says receiving E from pt; boolean expression holds only if it is true over the state. In the semantic definitions, $\sigma_1 \uplus \sigma_2$ represents the conjunction of two separated states, where σ_1 and σ_2 are required to have separated traces; $\sigma_1 \odot \sigma_2$ is the conjunction of sequential states which have sequentially connected traces. For the composite assertions, $p * q$ says p and q holds over separated states; $p \circ q$ holds over sequential states; p^* specifies traces that circulated for a finite times, where

$$\overline{p * q \Leftrightarrow q * p} \quad \overline{p \circ (q \circ r) \Leftrightarrow (p \circ q) \circ r} \quad \overline{(p * q) \circ r \Rightarrow (p \circ r) * q} \quad \overline{p \circ q \Rightarrow p * q}$$

$$\overline{p \circ (q * r) \Rightarrow (p \circ q) * r} \quad \overline{(r_1 * p_1) \circ (r_2 * p_2) \Rightarrow (r_1 \circ r_2) * (p_1 \circ p_2)}$$

$$\frac{\mathsf{Pure}(p) \text{ or } \mathsf{Pure}(q)}{p \circ q \Leftrightarrow p \wedge q} \quad \frac{\mathsf{Pure}(p) \text{ or } \mathsf{Pure}(q)}{p * q \Leftrightarrow p \wedge q}$$

$$\frac{\mathsf{Pure}(p)}{p \wedge (q \circ r) \Rightarrow (p \wedge q) \circ (p \wedge r)} \quad \frac{\mathsf{Pure}(r)}{p \circ (q \wedge r) \Leftrightarrow (p \circ q) \wedge r} \quad \frac{\mathsf{Pure}(q)}{(p \wedge q) \circ r \Leftrightarrow (p \circ r) \wedge q}$$

Figure 8: Selected Proof Rules

σ^* is the state that have a static store s and its trace has a sub-trace that repeatedly appears.

We also define pure assertions like in the Separation Logic. Syntactically, *pure assertions* do not contain any trace predicates.

Definition 3 (Pure Assertion) *Assertion p is* pure, $\mathsf{Pure}(p)$, iff *the validity of p does not rely on the state of the trace, i.e.,*

$$\textit{if } (s, tr) \models p, \textit{ then for all } tr', \ (s, tr') \models p.$$

Fig. 8 lists some selected proof rules, which are sound w.r.t. the semantics.

5 Inference System

In this section, we introduce our inference system, which is a separation-based system for reasoning distributed programs.

5.1 Syntactic Control of Well-formedness

The inference rules are given in Fig. 9 and Fig. 10. In order to avoid tedious side conditions, Syntactic Control of Interference (SCI) [18] is adopted here. There are two syntactic context: $\mathcal{O}_{\mathsf{var}}$ for variable context, and $\mathcal{O}_{\mathsf{port}}$ for port context.

$$\mathcal{O}_{\mathsf{var}} ::= x_1, x_2, x_3, \ldots \qquad \mathcal{O}_{\mathsf{port}} ::= pt_1, pt_2, pt_3, \ldots$$

$\mathcal{O}_{\mathsf{var}}$ denotes the ownership of a set of variables, and $\mathcal{O}_{\mathsf{var}}, \mathcal{O}'_{\mathsf{var}}$ is the conjunctive ownership of two separated sets of variables. $\mathcal{O}_{\mathsf{port}}$ is a set of port names, which specifies the access permissions of ports. If $pt \in \mathcal{O}_{\mathsf{port}}$, the current

agent can withdraw messages out of pt, and other agents can only send messages to pt. For simplicity, we consider full permissions. It is possible to extend this definition with fractional permissions [17] as well.

SCI specifies the well-formedness of expressions, assertions, and programs. A variable/expression/assertion is well-formed, if and only if all its free variables are within the scope of a syntactic context. We use $\mathcal{O}_{\mathsf{var}} \vdash x$ **Var**/E **Exp**/p **Assert** to represent them respectively. The well-formedness of a program (command), $\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash C$ **Comm**, can be defined by the following selected rules:

$$\frac{}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \textbf{skip Comm}} \quad \frac{\mathcal{O}_{\mathsf{var}} \vdash x \textbf{ Var} \quad \mathcal{O}_{\mathsf{var}} \vdash E \textbf{ Exp}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash x := E \textbf{ Comm}}$$

$$\frac{\mathcal{O}_{\mathsf{var}} \vdash x \textbf{ Var}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}}, pt \vdash x := \textbf{recv}\,(pt) \textbf{ Comm}} \quad \frac{\mathcal{O}_{\mathsf{var}} \vdash E \textbf{ Exp} \quad pt \notin \mathcal{O}_{\mathsf{port}}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \textbf{send}\,(E, pt) \textbf{ Comm}}$$

$$\frac{\mathcal{O}_{\mathsf{var}} \vdash B \textbf{ Exp} \quad \mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash C \textbf{ Comm}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \textbf{while } B \textbf{ do } C \textbf{ Comm}}$$

A well-formed assignment, $x := E$, requires the ownership of x; $x :=$ **recv** (pt) requires full permission of x and $pt \in \mathcal{O}_{\mathsf{port}}$; and well-formed receive action, **send** (x, pt) requires $pt \notin \mathcal{O}_{\mathsf{port}}$. That means a thread can only receive messages from the ports that it owns, or send messages that it doesn't own (the ports of others). We don't list all the rules, because other rules are straight forward.

A syntactic context can be weakened by extending $\mathcal{O}_{\mathsf{var}}$:

$$\frac{\mathcal{O}_{\mathsf{var}} \vdash E \textbf{ Exp}}{\mathcal{O}_{\mathsf{var}}, \mathcal{O}'_{\mathsf{var}} \vdash E \textbf{ Exp}} \quad \frac{\mathcal{O}_{\mathsf{var}} \vdash p \textbf{ Assert}}{\mathcal{O}_{\mathsf{var}}, \mathcal{O}'_{\mathsf{var}} \vdash p \textbf{ Assert}} \quad \frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash C \textbf{ Comm}}{\mathcal{O}_{\mathsf{var}}, \mathcal{O}'_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash C \textbf{ Comm}}$$

5.2 Program Specification

The specification of a message-passing program takes the form as:

$$\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r', q\}$$

It specifies the partial correctness of D: if D starts from a pre-state satisfying $r * p$, where r for the environmental state and p for the local state, then D will not abort, and when D terminates, if the environment satisfies r', the local state satisfies q. If D contains only one agent, e.g., $i : C$, the specification can be written as:

$$\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash_i \{r, p\}\, C\, \{r', q\}$$

$$\frac{x \in \mathcal{O}_{\mathsf{var}} \quad pt \in \mathcal{O}_{\mathsf{port}} \quad x \notin \mathsf{freeVar}(r) \cup \mathsf{freeVar}(p)}{\mathcal{O}_{\mathsf{var}}, X; \mathcal{O}_{\mathsf{port}} \vdash_i \{r, p\}\, x := \mathbf{recv}\,(pt)\, \{r, p \circ pt?X \wedge x = X\}} \text{ (RECV)}$$

$$\frac{x \in \mathcal{O}_{\mathsf{var}} \quad pt \notin \mathcal{O}_{\mathsf{port}}}{\mathcal{O}_{\mathsf{var}}, \mathcal{O}_{\mathsf{port}} \vdash_i \{r, p\}\, \mathbf{send}\,(x, pt)\, \{r, p \circ pt!x\}} \text{ (SEND)}$$

$$\frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash_i \{r, p\}\, C_1\, \{r', p'\} \quad \mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash_i \{r', p'\}\, C_2\, \{r'', p''\}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash_i \{r, p\}\, C_1; C_2\, \{r'', p''\}} \text{ (SEQ)}$$

$$\frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash_i \{r, (p \wedge B)\}\, C_1\, \{r', q\} \quad \mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash_i \{r, (p \wedge \neg B)\}\, C_2\, \{r', q\}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash_i \{r, p\}\, \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\, \{r', q\}} \text{ (IF)}$$

$$\frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash_i \{r^*, p^* \wedge B\}\, C\, \{r^*, p^*\}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash_i \{r^*, p^*\}\, \mathbf{while}\, B\, \mathbf{do}\, C\, \{r^*, p^* \wedge \neg B)\}} \text{ (WHILE)}$$

Figure 9: Selected Inference Rules — Basics

It is innovative that we syntactically separate the pre- and post-conditions into two parts: one assumption for the environment, and one specification for the local state. In the precondition, r is the environmental assumption before the execution, and p specifies the local pre-state. During execution, D may receive (send) messages from (to) the environment, so we can calculate the post-assumption of environment from D's local requirements. When D terminates, its environmental assumption becomes r' and local state becomes q. Clearly, the trace specified by r' and q will not be shorter than r and p.

Note that we always consider well-formed triples in this paper. That is, for all expressions, assertions and programs in a tuple, they are implicitly required to be well-formed.

5.3 Reasoning Rules

In Fig. 9, the rule (RECV) for receiving commands is straightforward, where $\mathsf{freeVar}(\bullet)$ returns the set of all free variables in $\bullet$. In this rule, variable X is a fresh logical variable to represent the message received by the current agent. No environmental assumption should be made at this stage. The rule for send event is similar. Note that no agent can send messages to itself ($pt \notin \mathcal{O}_{\mathsf{port}}$). Both (SEQ) for sequential composition and (IF) for conditional are trivial. (WHILE) is normal too, where each iteration should maintain the validity of loop invariant $\{r^*, p^*\}$. Other rules, which are used for structural reasoning, are defined in Fig. 10.

Definition 4 (Environmental-aided Implication) *Environmental-aided implication, written as* $p \overset{r}{\Rightarrow} p'$*, implies* p' *from* p *with an extra coupled trace*

$$\frac{p \overset{r}{\Rightarrow} p' \quad q' \overset{r'}{\Rightarrow} q \quad \mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p'\}\, D\, \{r', q'\}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r', q\}} \text{ (CONSEQ-A)}$$

$$\frac{r_1 \Rightarrow r \quad r_2 \Rightarrow r' \quad \mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r', q\}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r_1, p\}\, D\, \{r_2, q\}} \text{ (CONSEQ-B)}$$

$$\frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r_1, q_1\} \;\; \mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r_2, q_2\}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r_1 \vee r_2, q_1 \vee q_2\}} \text{ (DISJ)}$$

$$\frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r_1, q_1\} \;\; \mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r_2, q_2\}}{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r_1 \wedge r_2, q_1 \wedge q_2\}} \text{ (CONJ)}$$

$$\frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r', q\} \quad \mathcal{O}'_{\mathsf{var}} \vdash r'' \; \mathbf{Assert} \quad \mathsf{notInterfere}(r'', \mathcal{O}_{\mathsf{port}})}{\mathcal{O}_{\mathsf{var}}, \mathcal{O}'_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r * r'', p\}\, D\, \{r' * r'', q\}} \text{ (FRM-ENV)}$$

$$\frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r', q\} \quad \mathcal{O}'_{\mathsf{var}} \vdash p' \; \mathbf{Assert} \quad \mathsf{notInterfere}(p', \mathcal{O}_{\mathsf{port}})}{\mathcal{O}_{\mathsf{var}}, \mathcal{O}'_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p * p'\}\, D\, \{r', q * p'\}} \text{ (FRM-LOC)}$$

$$\frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r', q\} \quad \mathcal{O}'_{\mathsf{var}} \vdash r'' \; \mathbf{Assert} \quad r' \bowtie_{\mathcal{O}_{\mathsf{port}}} q}{\mathcal{O}_{\mathsf{var}}, \mathcal{O}'_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r' \circ r'', q\}} \text{ (FRM-BHD)}$$

$$\frac{\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r', q\} \quad \mathcal{O}'_{\mathsf{var}} \vdash r'', p' \; \mathbf{Assert} \quad r'' \bowtie_{\mathcal{O}_{\mathsf{port}}} p'}{\mathcal{O}_{\mathsf{var}}, \mathcal{O}'_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r'' \circ r, p' \circ p\}\, D\, \{r'' \circ r', p' \circ q\}} \text{ (FRM-AHD)}$$

$$\frac{\begin{array}{l}\mathcal{O}_{\mathsf{var}1}; \mathcal{O}_{\mathsf{port}_1} \vdash \{\mathsf{emp}_{\mathsf{tr}}, p_1\}\, D_1\, \{r_1, q_1\} \quad q_1 * r \Rightarrow r_2 * r'_2 \quad \mathsf{notInterfere}(r'_2, \mathcal{O}_{\mathsf{port}_2}) \\ \mathcal{O}_{\mathsf{var}2}; \mathcal{O}_{\mathsf{port}_2} \vdash \{\mathsf{emp}_{\mathsf{tr}}, p_2\}\, D_2\, \{r_2, q_2\} \quad q_2 * r \Rightarrow r_1 * r'_1 \quad \mathsf{notInterfere}(r'_1, \mathcal{O}_{\mathsf{port}_1})\end{array}}{\mathcal{O}_{\mathsf{var}1}, \mathcal{O}_{\mathsf{var}2}; \mathcal{O}_{\mathsf{port}_1}, \mathcal{O}_{\mathsf{port}_2} \vdash \{\mathsf{emp}_{\mathsf{tr}}, p_1 * p_2\}\, D_1 || D_2\, \{r, q_1 * q_2\}} \text{ (PAR)}$$

Figure 10: Selected Inference Rules — Others

r. It is true when $\forall \sigma_1 = (s, tr_1), \sigma_2 = (s, tr_2)$:

$$\begin{array}{c}\sigma_1 * \sigma_2 \models r * p \wedge \sigma_1 \models r \wedge \sigma_2 \models p \\ \wedge\, \forall e \in tr_2 \cdot \mathsf{isRecv}(e) \Rightarrow e.\mathsf{src} \in \mathsf{dom}(tr_1)\end{array} \Rightarrow \sigma_2 \models p'$$

Def. 4 enables local deduction ($p \Rightarrow q$) with the extra knowledge about environmental state (r). Particularly, all receives in the local state should match with some sends in the environment. For instance, we have $pt?X \overset{pt!2}{\Longrightarrow} X = 2$, while $pt?X * pt!2 \Rightarrow X = 2$ is not true since $*$ does not enforce send-receive matches in the trace.

In Fig. 10, there are two rules of consequences. (CONSEQ-A) adopts environmental aided implication for local deduction. To weaken a specification, we can either strengthen its local precondition, or weaken its local post-condition. (CONSEQ-B) is for the environmental state. Different from (CONSEQ-A), it only allows strengthening the assumption of environment, either at precondition or postcondition. These two rules are usually applied

together with rule (RECV). (RECV) makes no assumption about the message received. Using (CONSEQ-B), the current agent can make assumption for the received value by strengthening the predicate in the assumption part. Then, by using (CONSEQ-A), we can deduce the local state with the aid of a stronger environmental assumption.

Rules (DISJ) and (CONJ) allows combing bifurcated deductions. For (DISJ), the two post-assertions rely on different environmental assumptions, so if at least one of these assumptions is satisfied, the local state must satisfies $q_1 \vee q_2$. (CONJ) is similar that needs no more explanation.

Our system supports spatial modularity, because it allows the proof of a local agent to be extended with a frame by $*$, as long as the frame *does not interfere* with existing proofs.

Definition 5 (Non-Interference) *For an assertion r, we say that r does not interfere with $\mathcal{O}_{\mathsf{port}}$, written as* $\mathsf{notInterfere}(r, \mathcal{O}_{\mathsf{port}})$*, when:*

$$r \Rightarrow (\mathsf{true}_{\mathsf{tr}} * (pt!\_ \vee pt?\_) \Rightarrow pt \notin \mathcal{O}_{\mathsf{port}})$$

Predicate $\mathsf{notInterfere}(r, \mathcal{O}_{\mathsf{port}})$ says that the trace specified by r does not interfere with $\mathcal{O}_{\mathsf{port}}$, that is, it does not send or receive messages via any port in $\mathcal{O}_{\mathsf{port}}$.

The spatial modularity is described by rules (FRM-ENV) and (FRM-LOC) in Fig. 10. Rule (FRM-ENV) is the frame rule for the environment. It allows the environment to be extended with frame r'', as long as r'' does not interfere with D, i.e., r'' must not race with r' by sending messages to D; and not race with q by receiving messages form r'. Rule (FRM-LOC) is the frame rule for local state. If p' does not contain any message passing predicates, this rule is reduced to the standard frame rule in SL. If p' contains some message passing events, it must not interfere the existing communication between r' and q.

Definition 6 (Hooked Assertions) *Assertion p is* hooked with q *by $\mathcal{O}_{\mathsf{port}}$, which is denoted as $p \bowtie_{\mathcal{O}_{\mathsf{port}}} q$,* iff *for any trace tr such that $tr \models p * q$, any event $e \in \mathsf{dom}(tr)$, and any port $pt \in \mathcal{O}_{\mathsf{port}}$, the following conditions hold:*

$$\mathsf{isSend}(e) \wedge e.\mathsf{port} = pt \Rightarrow \exists e' \in \mathsf{dom}(tr) \cdot e'.\mathsf{src} = e, \ \textit{and}$$
$$\mathsf{isRecv}(e) \wedge e.\mathsf{port} = pt \Rightarrow \exists e' \in \mathsf{dom}(tr) \cdot e.\mathsf{src} = e'.$$

Intuitively, $p \bowtie_{\mathcal{O}_{\mathsf{port}}} q$ says that for any trace tr which satisfies $p * q$, there is no pending send or receive event that accesses ports in $\mathcal{O}_{\mathsf{port}}$.

Example 1 *Let* $\mathcal{O}_{\mathsf{port}} = \{pt\}$, $posiSend = (\exists X \cdot pt!X \wedge X > 0)^*$, $posiRecv = (\exists X \cdot pt?X \wedge X > 0)^*$, *we will have*

- $posiSend \bowtie_{\mathcal{O}_{\mathsf{port}}} posiRecv$ *does not hold, and*
- $(posiSend \circ pt!0) \bowtie_{\mathcal{O}_{\mathsf{port}}} (posiRecv \circ pt?0)$ *holds.*

Here $posiSend$ *is a sequence of events sending positive numbers, and* $posiRecv$ *is a sequence of events receiving positive numbers. These two assertions are not hooked, because there may exist pending events in* $posiSend$*. However, the second pair above is hooked, since each sequence is appended with a sentinel* 0 *at the end, which enforces each send to be paired with a receive and vice versa.*

There are some rules for hooked assertions, which are useful for program reasoning.

$$\frac{p_1 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_1 \quad p_2 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_2}{p_1 \circ p_2 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_1 \circ q_2} \qquad \frac{p_1 \circ p_2 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_1 \circ q_2 \quad p_1 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_1}{p_2 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_2}$$

$$\frac{p_1 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_1 \quad p_2 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_2}{p_1 * p_2 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_1 * q_2} \qquad \frac{p_1 * p_2 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_1 * q_2 \quad p_1 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_1}{p_2 \bowtie_{\mathcal{O}_{\mathsf{port}}} q_2}$$

With the definition of hooked assertion, we support temporal modularity as well, that is, we allow a trace to be connected ahead or behind of current trace. Temporal modularity is supported by rules (FRM-BHD) and (FRM-AHD). Rule (FRM-BHD) allows appending an extra r'' to the end of environmental assumption. To ensure soundness, r' should be hooked with q' so that the extra trace r'' in the environment should not affect the behavior of existing trace q. Rule (FRM-AHD) allows appending the frame $r'' * p'$ ahead of the current trace, where r'' is added to the environment, and p' is added to the local trace. The two assertions must be hooked together so that they do not affect the communication of later traces. This rule can be applied when proving sequential composition in programs.

Informally, in order to prove $C_1; C_2$, we can prove C_1 and C_2 independently in the first to get, e.g., $\{r_1, p_1\}\, C_1\, \{r_1', p_1'\}$ and $\{\mathsf{emp}_{\mathsf{tr}}, \mathsf{emp}_{\mathsf{tr}}\}\, C_2\, \{r_2', p_2'\}$. By applying (FRM-AHD), C_2 satisfies $\{r_1', p_1'\}\, C_2\, \{r_1' \circ r_2', p_1' \circ p_2'\}$. Therefore by rule (SEQ), the conjunct program can be specified by $\{r_1, p_1\}\, C_1; C_2\, \{r_1' \circ r_2', p_1' \circ p_2'\}$.

Rule (PAR) is for parallel composition of separated agents. For $D_1 || D_2$, the local trace of D_1 becomes the environment of D_2, and vice versa. In the rule, r is the environment of D_1 and D_2. Informally, D_1's environment is

$r * \text{traceOf}(D_2)$, and D_2's environment is $r * \text{traceOf}(D_1)$[6]. $q_1 * r \Rightarrow r_2 * r_2'$ ensures D_2's environment is satisfied; and $q_2 * r \Rightarrow r_1 * r_1'$ ensures D_1's environment is satisfied.

6 Semantics and Soundness

From the view of a local agent, state transitions could be caused either by itself or by other agents. We use $(D, \sigma) \overset{\lambda}{\rightsquigarrow} (D', \sigma')$ to represent the transition of a local agent, where $\lambda \in \{\mathbf{e}, \mathbf{l}\}$, $\mathbf{e}$ denotes the transition caused by the environment, and $\mathbf{l}$ denotes the transition caused by the agent itself.

Consider the configuration of an agent $(i : C, s, tr)$, where the domain of s is the set of variables owned by i. From its local view, the local state includes store s, and local event trace (a sub-state of tr), and environmental event trace (the other part of tr). *Local event trace* of agent i contains all events e in tr where $\mathsf{agent}(e) = i$.

Definition 7 (Trace Projection) *$tr \downharpoonright \{i_1, ..., i_m\}$ projects tr to the local trace of agents $\{i_1, ..., i_m\}$:*

$$(tr \downharpoonright \{i_1, ..., i_m\})(e) \stackrel{\text{def}}{=} \begin{cases} tr(e) & e \in \mathsf{dom}(tr) \wedge \mathsf{agent}(e) \in \{i_1, ..., i_m\} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Definition 8 (State Projection) *Let $D = i_1 : C_1 \,||\, \ldots \,||\, i_m : C_m$ be a distributed program, then for any state $\sigma = (s, tr)$, we define the local state and environmental state for D:*

$$\begin{array}{lcl} \sigma.\mathsf{loc} & \stackrel{\text{def}}{=} & (s, tr \downharpoonright \{i_1, \ldots, i_m\}) \\ \sigma.\mathsf{env} & \stackrel{\text{def}}{=} & \sigma' \quad \text{if} \quad \sigma = \sigma.\mathsf{loc} \uplus \sigma' \end{array}$$

State projection separates a state into local and environmental parts: $\sigma.\mathsf{loc}$ is the local state for agents in D; and $\sigma.\mathsf{env}$ is the environmental state.

Lemma 1 *For any transition $(D, \sigma) \overset{\lambda}{\rightsquigarrow} (D', \sigma')$:*

- *if $\lambda = \boldsymbol{e}$, then $D = D' \wedge \sigma.\mathsf{loc} = \sigma'.\mathsf{loc}$*
- *if $\lambda = \boldsymbol{l}$, then $\sigma.\mathsf{env} = \sigma'.\mathsf{env}$*

[6] traceOf(D) is an informal operator that returns the trace state of program D.

In Lemma 1, the first property says the environment could not affect the state of local resource; and the second property specifies that local state transition would not affect the current environmental state.

Definition 9 (Semantics) $\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \models \{r, p\}\, D\, \{r', q\}$ *holds, iff for all* σ *such that* $\sigma.\mathsf{env} \models r \wedge \sigma.\mathsf{loc} \models p$*, and if the followings are true:*

- $(D, \sigma) \overset{\lambda}{\rightsquigarrow}{}^{*} (\mathbf{skip}^{7}, \sigma')$*;*
- $\sigma'.\mathsf{env} \models r'$*.*

Then $\sigma'.\mathsf{loc} \models q$*.*

Intuitively, definition of $\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \models \{r, p\}\, D\, \{r', q\}$ says that D starts from a state $\sigma \models r * p$, where r specifies the state of environment, p specifies local state. Then if the D terminates at the state σ', that $\sigma'.\mathsf{env} \models r'$, then $\sigma'.\mathsf{loc} \models q$.

Theorem 3 (Soundness) *If the specification of* D *is* $\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \vdash \{r, p\}\, D\, \{r', q\}$*, then* $\mathcal{O}_{\mathsf{var}}; \mathcal{O}_{\mathsf{port}} \models \{r, p\}\, D\, \{r', q\}$*.*

The soundness is proved by induction over reasoning rules. By proving the soundness for each rule, the theorem follows by a straightforward rule induction. Detailed proof of this theorem is available online [12].

7 Non-determinism: A Case Study

The difficulty for understanding a concurrent program lies on non-deterministic inter-thread interleaving, for both shared memory and message-passing models.

In our setting, since each port belongs to one agent, all receive events on a port are therefore deterministically ordered by the syntactic program order. In this circumstance, non-determinism is only caused by send events to the same port which come from different agents, because these events are not ordered by the happens-before relation.

Consider the following example:

```
send (2, pt);   ||   send (3, pt);   ||   x := recv (pt);
                                          y := recv (pt);
```

[7] Here the ***skip*** represents that the code of each agent in D is the *skip*.

The program consists of two senders (two agents which send messages, namely sender_1 and sender_2) and one receiver (receiver). The final state of the receiver could be either $x = 2 \wedge y = 3$ or $x = 3 \wedge y = 2$, since the two sends do not have a determined order. This program can be proved by different ways based on associativity: either by $(\text{sender}_1 \,||\, \text{sender}_2) \,||\, \text{receiver}$, or by $\text{sender}_1 \,||\, (\text{sender}_2 \,||\, \text{receiver})$, or $\text{sender}_2 \,||\, (\text{sender}_1 \,||\, \text{receiver})$, where the later two possibilities are quite symmetric, thus we will consider only the first two cases below.

For the first case, the post-condition of two senders, $\text{sender}_1 \,||\, \text{sender}_2$, is:

1. $\{\mathsf{emp}_{\mathsf{tr}}, pt!2 * pt!3\}$

Note that the two send events are unordered, so they are connected by $*$.

For the receiver, the proof takes the similar form as the proof example in Section 1, we can prove the following two specifications:

2. $\{\mathsf{emp}_{\mathsf{tr}}, \mathsf{emp}_{\mathsf{tr}}\}$
   `x := recv (pt);`
   `y := recv (pt);`
3. $\{pt!2 \circ pt!3, x = 2 \wedge y = 3 \wedge \mathsf{true}_{\mathsf{tr}}\}$

4. $\{\mathsf{emp}_{\mathsf{tr}}, \mathsf{emp}_{\mathsf{tr}}\}$
   `x := recv (pt);`
   `y := recv (pt);`
5. $\{pt!3 \circ pt!2, x = 3 \wedge y = 2 \wedge \mathsf{true}_{\mathsf{tr}}\}$

Applying rule (DISJ) to the two deductions, the post-condition for the receiver is:

6. $\{pt!2 \circ pt!3 \vee pt!3 \circ pt!2, ((x = 2 \wedge y = 3) \vee (x = 3 \wedge y = 2)) \wedge \mathsf{true}_{\mathsf{tr}}\}$

Since $pt!2 * pt!3 \Rightarrow (pt!2 \circ pt!3 \vee pt!3 \circ pt!2)$, the environmental part of line 6 is satisfied by the local state of line 1. By parallelling line 1 and 6, we have:

7. $\{\mathsf{emp}_{\mathsf{tr}}, ((x = 2 \wedge y = 3) \vee (x = 3 \wedge y = 2)) \wedge \mathsf{true}_{\mathsf{tr}}\}$

Line 7 is the specification of the whole system. It depicts the two possible post-conditions for the program.

Another proof is based on the second associative form, which combines the receiver with a sender first. We present a proof for the receiver first:

8. $\{\mathsf{emp}_{\mathsf{tr}}, \mathsf{emp}_{\mathsf{tr}}\}$
9. $\{pt!2 * pt!3, \mathsf{emp}_{\mathsf{tr}}\}$
   `x := recv (pt);`
   `y := recv (pt);`
10. $\{pt!2 * pt!3, pt?X \circ pt?Y \wedge x = X \wedge y = Y\}$
11. $\{pt!2 * pt!3, ((x = 2 \wedge y = 3) \vee (x = 3 \wedge y = 2)) \wedge \mathsf{true}_{\mathsf{tr}}\}$

Line 9 is obtained by line 8 by assuming two unordered send events in the environmental state. Line 11 comes from line 10 according to environmental aided deduction: there are two possible post-conditions for the value of x and y.

The post-condition of one sender is $\{\mathsf{emp}_{\mathsf{tr}}, pt!3\}$. According to rule (PAR), the sender needs to conjoin with another environmental assumption, which is $pt!2$ here, in order to satisfy the environmental state of line 11. By rule (PAR), we can parallel compose the sender with the receiver and have a resulted post-condition:

$$\texttt{12.}\quad \{pt!2, ((x = 2 \wedge y = 3) \vee (x = 3 \wedge y = 2)) \wedge \mathsf{true}_{\mathsf{tr}}\}$$

The other sender can satisfy the rest of the assumption in line 12, which results in the same post-condition as line 7. This step is similar and thus omitted.

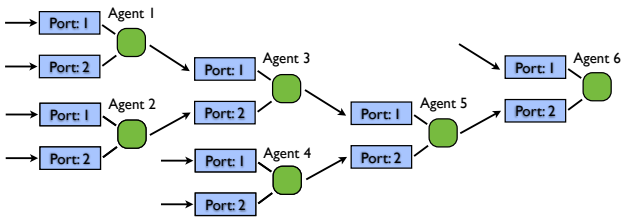
The examples shows the usage of our logic to reason the systems that exhibit non-deterministic behaviors. The local thread only needs to assume unordered send events in the environment, which will result in non-deterministic local state from environmental aided deduction. The proofs for a program can be various depending on different association among threads.

8 Example: Filters

Filters form a common class of distributed systems. A filter is an agent that receives messages from one or more ports and send messages to some other ports. In this section, we prove a filter example — Merging Network.

Fig. 11 (upper part) shows the architecture of a merging network. Each agent in the network is a filter that merges two monotonic positive streams into one monotonic stream, and 0 marks the end of streams. Fig. 11 (lower part) shows an implementation of agent 5. Here each port takes a unique ID, and $k@i$ denotes port k of agent i.

Agent 5 owns two variables and two ports, and is sequentially composed by three while loops. As the proof of the trivial example in Section 1, we prove these loops separately and then sequentially compose these independent proofs together.




```
agent5 () {
  v1 = recv (1@5); v2 = recv (2@5);
  while (v1 ≠ 0 ∧ v2 ≠ 0){
    if (v1 > v2) {send (v2, 2@6); v2 := recv (2@5);}
    else {send (v1, 2@6); v1 := recv (1@5);}}
  while (v1 ≠ 0) {send (v1, 2@6); v1 := recv (2@5);}
  while (v2 ≠ 0) {send (v2, 2@6); v2 := recv (1@5);}
  send (0, 2@6);
}
```

Figure 11: Merge Sort

For clarity, we define the following predicates to simplify descriptions:

$$\begin{aligned}
\mathsf{mono}(pt) &\stackrel{\text{def}}{=} (\mathsf{true}_{\mathsf{tr}} * pt!X) \circ (\mathsf{true}_{\mathsf{tr}} * pt!Y) \Rightarrow 0 < X \leq Y \\
\mathsf{monoEnd}(pt) &\stackrel{\text{def}}{=} \mathsf{mono}(pt) \circ pt!0 \\
\mathsf{large}(var) &\stackrel{\text{def}}{=} \forall n \cdot (\mathsf{true}_{\mathsf{tr}} * 2@6!n \Rightarrow var = 0 \vee var \geq n) \\
\mathsf{eqLast}(pt, var) &\stackrel{\text{def}}{=} \mathsf{true}_{\mathsf{tr}} \circ pt?X \Rightarrow var = X
\end{aligned}$$

$\mathsf{mono}(pt)$ says the environment sends positive monotonic messages to pt, and $\mathsf{monoEnd}(pt)$ additionally says the stream has ended; $\mathsf{large}(var)$ says var is either larger than any message that previously sent to 2@6 or equal to 0; $\mathsf{eqlast}(pt, var)$ says var equals to the last message that received from pt.

Fig. 12 lists the proof for the first loop. Line 1 is the loop invariant. It assumes that the environment send monotonic streams to 1@5 and 2@5, and its local trace is a composition of some receive events of that two ports and a set of monotonic sends to port 2@6. Line 2 is obtained from line 1 by conjoining the boolean condition of the while. Line 3 falls into a branch of the if statement. In this branch, the agent sends v_2 and receives messages from 2@5. Therefore, we treat some assertions about v_1 and 1@5 in line 2 as frame, and line 3 is deduced by framing out those irrelevant traces. Line 4 is the same as line 3, because according to $\mathsf{large}(v_2) \wedge v_2 \neq 0$, v_2 is larger than any message that was previously sent to 2@6, so $\mathsf{mono}(2@6)$ holds in line 4; also since $v_1 > v_2$, $\mathsf{large}(v_1)$ remains to be true in line 4. The deduction from line 4 to 5 is also the standard local deduction. Deductions for the

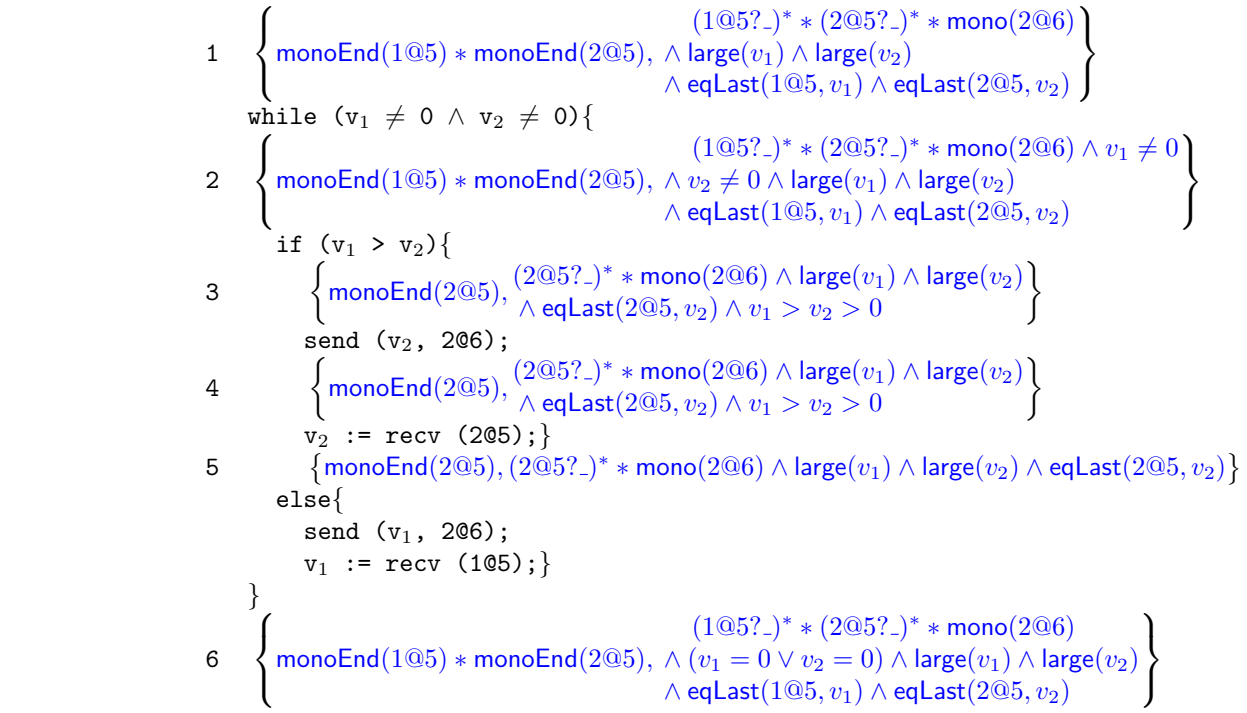


Figure 12: Proof of the First While-loop

other branch are symmetric and thus omitted. Line 6 is obtained from line 5 by conjoining with the frame that was put aside in line 3.

Proof of the second loop is given in Fig 13. Line 7 is obtained from line 6 by framing out irrelevant assertions about 2@5 and adjoining the boolean predicate that guarded by the loop. The proof from line 7 to line 8 is just local reasoning as the proof from line 3 to 5. Line 9 is obtained by conjoining the frame of line 7 back to the post-condition. The third loop is symmetric with the second, and therefore we omit its proof.

The sketch of the overall proof is given in Fig. 14, where we only present the assertions at the critical places, e.g., the position where a framework is put aside or taken back. As we have already discussed, the overall proof is obtained by sequentially conjoining several separated proofs of some locally connected commands. The post-condition says that if the environment send monotonic streams to the two ports of agent 5, then the agent will send monotonic streams to 2@6.

Note that the specification of agent 5 can be joined with the specification

```
   while (v1 ≠ 0){
7    {monoEnd(1@5), (1@5?_)* ∗ mono(2@6) ∧ v1 ≠ 0 ∧ v2 = 0
                    ∧ large(v1) ∧ eqLast(1@5, v1)}
     send (v1, 2@6);
     v1 := recv (2@5);
8    {monoEnd(1@5), (1@5?_)* ∗ mono(2@6) ∧ v2 = 0 ∧ large(v1)
                    ∧ eqLast(1@5, v1)}
   }
9  {monoEnd(1@5) ∗ monoEnd(2@5), (1@5?_)* ∗ (2@5?_)* ∗ mono(2@6)
                                ∧ v1 = 0 ∧ v2 = 0}
```

Figure 13: Proof of the Second While-loop

of other agents, just like the proof of the tiny example in Section 1. It is feasible if other agents take different algorithms as long as the assumption of agent 5 is satisfied.

9 Example: 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_i 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.

Since each agent has only one port, we reuse agent ID as port ID. The program is given in Fig. 15. 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 $\mathsf{maxtk}(i, j)$ 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:

$$p(i) \stackrel{\text{def}}{=} \forall j \cdot (i+1)!\mathsf{token}_j \Rightarrow \mathsf{token}_j = \mathsf{maxtk}(j, i)$$

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

```
agent5 (){
  {emp_tr, emp_tr}
  v1 = recv (1@5);
  v2 = recv (2@5);
  {1@5!X * 2@5!Y, 1@5?X ∘ 2@5?Y ∧ v1 = X ∧ v2 = Y}
  {monoEnd(1@5) * monoEnd(2@5), 1@5?X ∘ 2@5?Y ∧ v1 = X ∧ v2 = Y}
  { monoEnd(1@5) * monoEnd(2@5), (1@5?_)* * (2@5?_)* * mono(2@6)
                                 ∧ large(v1) ∧ large(v2)
                                 ∧ eqLast(1@5, v1) ∧ eqLast(2@5, v2) }
  while (v1 ≠ 0 ∧ v2 ≠ 0){
    if (v1 > v2){send (v2, 2@6); v2 := recv (2@5);}
    else{send (v1, 2@6); v1 := recv (1@5);}
  }
  { monoEnd(1@5) * monoEnd(2@5), (1@5?_)* * (2@5?_)* * mono(2@6)
                                 ∧ (v1 = 0 ∨ v2 = 0) ∧ large(v1) ∧ large(v2)
                                 ∧ eqLast(1@5, v1) ∧ eqLast(2@5, v2) }
  while (v1 ≠ 0){
    { monoEnd(1@5), (1@5?_)* * mono(2@6) ∧ v1 ≠ 0 ∧ v2 = 0
                    ∧ large(v1) ∧ eqLast(1@5, v1) }
    send (v1, 2@6); v1 := recv (2@5);
    {monoEnd(1@5), (1@5?_)* * mono(2@6) ∧ v2 = 0 ∧ large(v1) ∧ eqLast(1@5, v1)}
  }
  while (v2 ≠ 0){send (v2, 2@6); v2 := recv (1@5);}
  {monoEnd(1@5) * monoEnd(2@5), (1@5?_)* * (2@5?_)* * mono(2@6) ∧ v1 = 0 ∧ v2 = 0}
  send (0, 2@6);
  {monoEnd(1@5) * monoEnd(2@5), (1@5?_)* * (2@5?_)* * monoEnd(2@6)}
}
```

Figure 14: Proof Sketch of Merge Sort

We are expected to prove that for any agent i, if the local variable $ld_i = \mathrm{tt}$, then agent i holds the largest token around the ring. We use predicate $\mathsf{valid}(ld_i)$ to specify the status of ld_i:

$$\mathsf{valid}(ld_i) \stackrel{\text{def}}{=} ld_i = \mathrm{tt} \Rightarrow \mathsf{token}_i = \mathsf{maxtk}(1, n)$$

The system should satisfy the following tuple:

$$\{\mathsf{emp}_{\mathsf{tr}}, \forall i \cdot \mathsf{valid}(ld_i)\}$$

Note that $\mathsf{valid}(ld_i)$ always holds when $ld_i = \mathsf{ff}$.

Proof of one agent in the program is given in Fig. 16, where we have proved the following triple:

$$\{\mathsf{emp}_{\mathsf{tr}}, \mathsf{emp}_{\mathsf{tr}}\}\ agent_{\mathtt{i}}\ \{p(i-1), p(i) \wedge valid(ld_i)\}$$

```
agent_i(){
  ld_i := ff;
  send (token_i, i + 1);
  tk_i := recv (i − 1);
  while (tk_i ≠ 0){
    if (token_i < tk_i){send (tk_i, i + 1);}
    if (token_i = tk_i){ld_i = tt; send (0, i + 1);}
    tk_i := recv (i-1);
  }
  if (ld_i = ff){send (0, i+1);}
}
```

Figure 15: Leader Election: Program

From the triple, we proved that the local state of each agent satisfies the environmental assumption of the next agent. Since all agents are connected in a circle, so the system is self-satisfied, and we can easily have the following specification using (PAR) rule:

$$\{\mathsf{emp}_{\mathsf{tr}}, \mathsf{emp}_{\mathsf{tr}}\}\, agent_1 \,||\, \ldots \,||\, agent_{\mathtt{n}}\, \{\mathsf{emp}_{\mathsf{tr}}, \mathsf{valid}(ld_1) \wedge \ldots \wedge \mathsf{valid}(ld_n)\}$$

Therefore, in the post-condition, each thread in the system holds a valid boolean. If a boolean is tt, the agent must have the largest token around the circle.

Leader election is a classic algorithm that has been proved by many others. However, the most distinguished feature is that our method can directly prove upon the real code, rather than mathematical abstractions and auxiliary lemmas. Besides, the system is proved modularly: each agent is proved against its local environmental reliance, and the parallel composition ensures the system to be self-contained.

10 Related Work and Conclusions

Now we summarize some related work, and then give a conclusion.

Program verification has been studied from various standpoints for decades. Verification of concurrent systems is especially interesting because of their inherent non-determinism.

Process Calculus. For the message-passing models, there exist many well-known works on process calculi, e.g., CSP (Communicating Sequential

```
agent_i(){
  {emp_tr, emp_tr}
  ld_i := ff;
  {emp_tr, emp_tr ∧ ld_i = ff}
  {emp_tr, emp_tr ∧ valid(ld_i)}
  send (token_i, i + 1);
  {emp_tr, (i + 1)!token_i ∧ valid(ld_i)}
  tk_i := recv (i − 1);
  {emp_tr, (i + 1)!token_i ∘ (i − 1)?X ∧ tk_i = X ∧ valid(ld_i)}
  {p(i − 1), p(i) ∧ (true_tr ∘ (i − 1)?X) ∧ tk_i = X ∧ valid(ld_i)}
  while (tk_i ≠ 0){
    {p(i − 1), p(i) ∧ (true_tr ∘ (i − 1)?X) ∧ tk_i = X ∧ X ≠ 0 ∧ valid(ld_i)}
    if (token_i < tk_i){
      {p(i − 1), p(i) ∧ (true_tr ∘ (i − 1)?X) ∧ tk_i = X ∧ X > token_i ∧ valid(ld_i)}
      send (tk_i, i + 1);
      {p(i − 1), p(i) ∧ valid(ld_i)}
    }
    if (token_i = tk_i){
      {p(i − 1), p(i) ∧ (true_tr ∘ (i − 1)?X) ∧ tk_i = X ∧ X = token_i ∧ valid(ld_i)}
      {p(i − 1), p(i) ∧ tk_i = token_i ∧ token_i = maxtk(i + 1, i)}
      ld_i = tt;
      {p(i − 1), p(i) ∧ tk_i = token_i ∧ token_i = maxtk(i + 1, i) ∧ ld_i = tt}
      {p(i − 1), p(i) ∧ valid(ld_i)}
      send (0, i + 1);
      {p(i − 1), p(i) ∧ valid(ld_i)}
    }
    tk_i := recv (i-1);
    {p(i − 1), p(i) ∧ (true_tr ∘ (i − 1)?X) ∧ tk_i = X ∧ valid(ld_i)}
  }
  {p(i − 1), p(i) ∧ (true_tr ∘ (i − 1)?0) ∧ tk_i = 0 ∧ valid(ld_i)}
  if (ld_i = ff){send (0, i+1);}
  {p(i − 1), p(i) ∧ valid(ld_i)}
}
```

Figure 16: Leader Election: Proof

Processes) [7], CCS (Calculus of Communicating System) [13], π-calculus [15], and KPN (Kahn Process Network) [9]. However, those algebraic systems focus mainly on the agent behavior deductions and equivalence, e.g., bi-simulation. It is not very clear how to apply those calculi to modularly specify and reason the properties of local states of the agents, which are written in real code and defined with stated-based semantics. The later is

the main focus in this work.

Separation Logic. Recently there is a clear trend that concurrency verification should support better modularity and locality, in order to verify larger systems. Modular verification of shared memory models has gained much progress since the development of the Separation Logic [19]. Separation Logic treats the program state as *resource*, and modularity is achieved by curving irrelevant resource (namely the *frame*) out of current state and conjoining the frame back when merging the local state back into the environment. In considering shared memory concurrency, a typical idea is to take the state of memory (*heap*) as the resource, which is a mapping from locations to values. In the reasoning, a portion of memory could be curved out or merged back, as well as transferred between the agents, those make the inference modular.

However, in the message-passing model, event traces are, unlike heap, well-organized structures that associate with many add-on restrictions, e.g., acyclic, send-receive match, *etc.* The complexity of trace structures impedes state-based Hoare type reasoning. The challenge (and also a shining spot of our paper) is to structurally specify event traces so that local reasoning could be achieved by curving out irrelevant events, as in other Separation Logic related works. Our framework solves this problem by introducing two operators to depict the separation of traces, so that the traces could be either separately connected or temporally connected; and introducing four frame rules, so that frames could be added in four ways: adding in environmental or local trace, or adding ahead or behind of the current trace. These make our system flexible and powerful.

Rely-Guarantee Based Reasoning. Rely-Guarantee (RG) reasoning was developed in 1980th [8]. In RG, a pair of Rely and Guarantee conditions are used to regulate the behaviors of the portions of the verified programs. The rely condition specifies the agents' local assumption on the environmental interferences, and guarantee condition is agents' interference upon other agents which form its environment. There are some limitations of the regular RG reasoning. In the first, the RG conditions are global and permanent, this requires a detailed understanding of the program to be proved prior any deduction. Second, the stability requirement of pre/post-conditions will also inhibit its accessibility.

RG reasoning has been extended with Separation Logic for verifying concurrent systems towards better modularity. The notable works in this

direction include Vafeiadis *et al.* [20] and Wehrman *et al.* [22]. Many interesting concurrent programs have been verified that show the power of the marriage of the RG reasoning with the Separation Logic. However, these works all target the shared memory model.

Our system gets clearly some ideas from RG. However, comparing with the regular RG reasoning (that applies also to the most recent works in this direction), ours has some innovations: (1) The rely condition of RG should be pre-defined and fixed, here we can dynamically determine, calculate and alter the environmental assumption. (2) Our environmental assumption is hidden once it is satisfied by other agents, rather than remained permanently as RG reasoning. This makes better modularity in the reasoning. (3) In RG reasoning, pre- and post-conditions are required to be *stable*, i.e., the assertions should remain valid no matter how environment interferes. Stability is a rather strong requirement that requires thoughtful assertion definitions. In our system, this requirement is also eliminated, that makes the reasoning process easier.

Other Works. W. de Roever *et al.* [4] published a book which made an excellent summarization with good coverage of previous works on the state-based verification of concurrent programs. The leading theme of the book is compositional techniques for concurrency verification. The book makes a comprehensive discussion about verification of both shared memory and message-passing models, and clearly, our work can be viewed as a new development on the same theme. However, there are some fundamental differences and contributions that distinguish our logic from W. de Roever *et al.*'s and many others: (1) by our limited knowledge, although Lamport's trace semantics has been proposed for decades, there is no state-based reasoning system defined based on this semantics; (2) the two-dimensional (temporal and spatial) modularity of our work, which is a benefit extracted from Lamport's semantics, has not been clearly touched by others; (3) our logic directly reason about imperative programming language, rather than some high level mathematical descriptions.

There are also many other works aiming at specifying and reasoning message-passing programs based on trace semantics. For instance, Bickford *et al.* [1] formally defined the event trace structures, and gave a minimal set of axioms for trace reasoning. Comparing with other trace-based reasoning, including Bickford', ours supports better modularity, and allows directly reasoning over existing code modules and conjoining separated proofs based

on several explicit conditions.

Villard *et al.* [21] proposed a separation-based logic for copyless message-passing models. One feature of their work is the support of ownership transfer. It is possible to extend our logic for ownership transfer, e.g., by adding the notion of "resource" for each agent and specifying those transmitted resource inside environmental assumptions. However, this solution is similar with Concurrent Separation Logic [2], and will not be able to provide much theoretical innovation. In another aspect, Villard's method is still defined based on the shared memory model, while ours is for pure message-passing systems.

Conclusion and future work.

We propose a compositional reasoning system for verifying distributed programs with asynchronous message passing. The work inherits and integrates some ideas from *Separation Logic*, *Rely-Guarantee reasoning*, and others, and archives very good modularity in both specification and verification. This reasoning framework exhibits two major contributions: First, we embody the concept of *event graphs* for distributed systems, and supports modular specification at both temporal and spatial dimensions. Second, we propose an innovative Hoare triple, which syntactically separates environmental and local assertions, to better specify and reason interactions between agents and the environment. We have applied this method to reason about some non-deterministic message-passing programs. The formal specification and verification of a filter network and a leader election algorithms are presented this paper. In the future, we will further test its applicability with more applications. It is also interesting to explore the possibility of building tools to automate the verification process.

References

[1] Mark Bickford and Robert L. Constable. A causal logic of events in formalized computational type theory. Technical report, Cornell University, 2005.

[2] Stephen D. Brookes. A semantics for concurrent separation logic. In Philippa Gardner and Nobuko Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR 2004)*, vol-

ume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004. doi:10.1007/978-3-540-28644-8_2.

[3] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, 1996. doi:10.1007/s004460050018.

[4] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods*. Cambridge University Press, January 2012.

[5] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer, 2010. doi:10.1007/978-3-642-14107-2_24.

[6] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In Rocco De Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007. doi:10.1007/978-3-540-71316-6_13.

[7] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.

[8] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and System*, 5(4):596–619, 1983. doi:10.1145/69575.69577.

[9] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

[11] Jinjiang Lei and Zongyan Qiu. Modular reasoning for message-passing programs. In Gabriel Ciobanu and Dominique Méry, editors, *Proceedings*

of the 11th International Colloquium on Theoretical Aspects of Computing (ICTAC 2014), volume 8687 of *Lecture Notes in Computer Science*, pages 277–294. Springer, 2014. doi:10.1007/978-3-319-10882-7_17.

[12] Jinjiang Lei and Zongyan Qiu. Modular reasoning for message-passing programs. Technical report, School of Mathematical Sciences, Peking University, Sep 2014. Available from: https://sites.google.com/site/jinjianglei/publications/rgdsep_journal.

[13] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.

[14] Robin Milner. *Communication and Concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[15] Robin Milner. The polyadic pi-calculus (abstract). In Rance Cleaveland, editor, *Proceedings of the Third International Conference on Concurrency Theory (CONCUR 1992)*, volume 630 of *Lecture Notes in Computer Science*, page 1. Springer, 1992. doi:10.1007/BFb0084778.

[16] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[17] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS 2006)*, pages 137–146. IEEE Computer Society, 2006. doi:10.1109/LICS.2006.52.

[18] Uday S. Reddy and John C. Reynolds. Syntactic control of interference for separation logic. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 323–336. ACM, 2012. doi:10.1145/2103656.2103695.

[19] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

[20] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR 2007)*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007. doi:10.1007/978-3-540-74407-8_18.

[21] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In Zhenjiang Hu, editor, *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS'09)*, volume 5904 of *Lecture Notes in Computer Science*, pages 194–209. Springer, 2009. doi:10.1007/978-3-642-10672-9_15.

[22] Ian Wehrman, C. A. R. Hoare, and Peter W. O'Hearn. Graphical models of separation logic. *Information Processing Letters*, 109(17):1001–1004, 2009. doi:10.1016/j.ipl.2009.06.003.