SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

December 2014, Vol. 57 128101:1–128101:18
doi: 10.1007/s11432-014-5150-4

A formal semantics for debugging synchronous message passing-based concurrent programs

LI He*, LUO Jie & LI Wei

State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing 100191*, China*

Received April 28, 2014; accepted June 9, 2014

Abstract In this paper, we propose a semantic framework to debug synchronous message passing-based concurrent programs, which are increasingly useful as parallel computing and distributed systems become more and more pervasive. We first design a concurrent programming language model to uniformly represent existing concurrent programming languages. Compared to sequential programming languages, this model contains communication statements, i.e., sending and receiving statements, and a concurrent structure to represent communication and concurrency. We then propose a debugging process consisting of a tracing and a locating procedure. The tracing procedure re-executes a program with a failed test case and uses specially designed data structures to collect useful execution information for locating bugs. We provide for the tracing procedure a structural operational semantics to represent synchronous communication and concurrency. The locating procedure backward locates the ill-designed statement by using information obtained in the tracing procedure, generates a fix equation, and tries to fix the bug by solving the fix equation. We also propose a structural operational semantics for the locating procedure. We supply two examples to test our proposed operational semantics.

Keywords debugging, synchronous, message passing, concurrent program, operational semantics

Citation Li H, Luo J, Li W. A formal semantics for debugging synchronous message passing-based concurrent programs. Sci China Inf Sci, 2014, 57: 128101(18), doi: 10.1007/s11432-014-5150-4

1 Introduction

Message passing-based concurrent programs are widely used in the field of parallel computing and distributed systems in order to handle growing amounts of data in different domains, such as mobile Internet, weather forecasting and image processing. Therefore, effectively debugging these kinds of concurrent programs with the aim of troubleshooting the various problems encountered has become very important nowadays.

Program debugging is a time-consuming process in software development because one has to trace the execution of a program with a failed test case and locate the bug by repeatedly narrowing the bounds of the anomaly. According to [1,2], debugging and testing of programs contribute to 50%–75% of the total development cost, whereas locating and understanding the nature of the errors usually amount to 90% of the debugging work.

* Corresponding author (email: lihe@nlsde.buaa.edu.cn)

Despite the availability of many techniques for modeling, testing, and debugging message passing-based concurrent programs, such as mathematical models [3–8], structural testing [9–12], reachability testing [13], replay [14–16], and monitoring execution information [17,18], debugging message passing-based concurrent programs is still extremely time-consuming, and the efficiency of the debugging relies heavily on the insight and experience of programmers.

A sequential program debugging framework designed using the notations and techniques of operational semantics [19] has recently been proposed in [20]. It enables a calculus to deal with the inconsistency between the actual result of executing a program and the expected result derived from its specification. Inspired by the formal semantics in [20], we design in this paper a semantic framework to debug synchronous message passing-based concurrent programs[1)]. The important aspects of the proposed framework are as follows:

• We design a concurrent programming language model according to three widely used real message passing-based programming languages. Compared to sequential programming languages, this model contains communication statements, i.e., sending and receiving statements, to represent communication as well as concurrent structure to represent concurrency. Synchronization is represented by the semantics for the communication statements.

• We provide a framework for debugging synchronous message passing-based concurrent programs, which contains a tracing procedure and a locating procedure. The tracing procedure re-executes the program with a failed test case to collect useful execution information. During this process, some specific data structures are used to record this information. The locating procedure locates the ill-designed statement by comparing the execution information obtained from the tracing procedure with the expected information specified by the specification. Finally, a fix equation is generated, and the locating procedure tries to fix the bug by solving the fix equation.

• We propose a structural operational semantics for the tracing procedure and the locating procedure, which are both represented by some rules. The rules for the tracing procedure carefully handle the semantics of concurrency and synchronous communication. The rules for the locating procedure contain step-by-step instructions for locating ill-designed statements.

The rest of the paper is organized as follows. In Section 2, we present three synchronous message passing-based concurrent programs and design a concurrent programming language model to represent the features common to different languages, i.e., concurrency and synchronous communication. In Sections 3 and 4, we provide the structural operational semantics for the tracing procedure and the locating procedure, respectively, along with a few examples to explicate the use of the proposed operational semantics. Related research is introduced in Section 5. We present our conclusions and directions for future work in Section 6.

2 Concurrent programming language model and basic notations

In the field of concurrent programming, there are many widely used message passing-based programming languages or libraries. Some of these are completely based on synchronous communication, such as Occam and Java Communicating Sequential Processes (JCSP). Others presume both synchronous and asynchronous communication, such as the Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). A program written by the synchronous parts of these languages is called a synchronous message passing-based concurrent program, or SyncCP for short in what follows. A SyncCP usually consists of several processes that can exchange messages through channels. A process that sends a message to a channel is called a sending process and the corresponding statement is called a sending statement. A process that receives a message from a channel is called a receiving process and the corresponding statement is called a receiving statement.

Figure 1 shows programs written in Occam, MPI and JCSP. Each program has two processes, the sending process and the receiving process. All three programs implement the same function, i.e., the

1) In this paper, we only address the situation where the program halts but the output is not as expected.

SyncCP$_1$ (by Occam)

```
/* Communication channel */
CHAN OF INT comm

PAR  /* Concurrent operator */

WHILE TRUE
INT val1:
SEQ  /* Sequential operator */
  Val1 = 2
  comm ! val1

WHILE TRUE
INT val2:
SEQ
  comm ? val2
```

(a)

SyncCP$_2$ (by MPI)

```
#include "mpi.h"
#define SIZE 10  /* Size of sending/receiving buffer */
int main( int argc, char **argv){
   int buffer[SIZE];
   act_size = 1;
   buf[0] = 2;
   MPI_Init(&argc, &argv);
    /* Get the process Id rank from communication world*/
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   if (rank ==  src) {
      /* The current process is sending process and sends
            the first act_size elements in sending buffer  */
      MPI_Ssend (buffer, act_size , dest, ……);
   }

   else if (rank ==  dest) {
      /*The current process is receiving process and receives
            the first act_size elements to receiving buffer. */
      MPI_Recv(buffer, act_size, src, ……);
   }
}
```

(b)

SyncCP$_3$ (by JCSP)

```
public class SendEvenIntsProcess implements CSProcess {
  private ChannelOutput out;
  ……     /* Constructor initialize the channel out */
  public void run(){
    i = 2;
    out.write (new Integer (i));
  }
}
public class ReadEvenIntsProcess implements CSProcess{
  private ChannelInput in;
  ……     /* Constructor initialize the channel in */
  public void run(){
    Integer d = ( Integer)in.read();
    System.out.println("Read: " + d.intValue());
  }
}
public class DriverProgram{
  One2OneChannel chan = new One2OneChannel();
  new CSProcess[]{
    /* The same channel */
    new SendEvenIntsProcess (chan),
    new ReadEvenIntsProcess (chan)
  }
}
```

(c)

Figure 1 Three synchronous message passing-based concurrent programs. (a) Written by Occam; (b) written by MPI; (c) written by JCSP.

sending process sends the value 2 to the receiving process.

• The Occam program uses the keyword **PAR** to create two processes. These processes transfer values through the channel **comm**, i.e., the sending process sends the value stored in **val1** to **comm** through the statement **comm!val1**, and the receiving process receives the value from **comm** and assigns the received value to **val2** through the statement **comm?val2**.

• The MPI program defines the communication world, where all processes in the same MPI concurrent program belong to the same communication world. The process id, which is used to distinguish processes, is obtained through **MPI_Comm_rank**. The sending process has a sending buffer and sends the data in the buffer to the target process (receiving process) through **MPI_Ssend**. The receiving process has a receiving buffer. It receives data from the source process (sending process) and stores it in the receiving buffer through **MPI_Recv**. Although the channel is not explicitly defined, the implementation of the MPI guarantees that there is a channel connecting the sending process and the receiving process when communication occurs.

• The JCSP program defines processes as objects, such as **SendEvenIntsProcess** and **ReadEvenIntsProcess**. The main function spawns two parallel processes by creating one sending object and one receiving object. The sending object contains a channel **out**, which means that the sending object will write values to this channel. The receiving object contains a channel **in**, which means that the it will read values from this channel. The main function passes the same channel **chan** to the sending process (sending object) and the receiving process (receiving object) so that these two processes can communicate through **chan**.

These real programs show that in addition to statements used in sequential programs, a concurrent program should contain syntax objects that are used for: 1) **Sending**, e.g., **comm!val1** in Occam, **MPI_Ssend** in MPI, and **out.write** in JSCP; 2) **Receiving**, e.g., **comm?val2** in Occam, **MPI_Recv** in MPI, and **in.read** in JCSP; 3) **Connecting**, e.g., **comm** in Occam, **chan** in JCSP, and the underlying implementation in MPI.

We design a concurrent programming language model to represent the important features common to these different languages. This model consists of four kinds of syntax objects: arithmetic expression e, boolean expression b, statement S, and concurrent program $Proc$. e and b are defined as usual. S and

Proc are defined as follows.

$$S ::= \textbf{skip} \mid x := e \mid \{S\} \mid \textbf{if}\, b\, \textbf{then}\, \{S_1\}\, \textbf{else}\, \{S_2\} \mid \textbf{while}\, b\, \textbf{do}\, \{S\} \mid S_1; S_2 \mid C!e \mid C?x,$$
$$Proc ::= pid : S \mid pid : S \,||\, Proc.$$

Here, $pid \in N^+$ is the process id used to uniquely represent a process, and C is a string used to represent the communication channel.

In comparison with sequential programs, the sending statement $C!e$ and receiving statement $C?x$ are added to statement S to represent communication. $C!e$ means that the sending process sends the value of expression e to C, whereas $C?x$ means that the receiving process receives the value from C and assigns the received value to variable x. Communication channel C is unidirectional, which means that each channel has a fixed direction of communication and connects exactly two processes.

Proc represents the concurrent program SyncCP. It usually consists of several processes separated by the parallel operator ||. In particular, a concurrent program can only contain a single process $pid : S$, which can be regarded as a sequential program. Processes in Proc shall be executed in parallel and may communicate with each other during execution.

In the following, we give some basic definitions and notations. "State" is an abstract description of a configuration of the memory and is defined as follows:

Definition 1 (State)**.** A state σ is a map $\sigma : V \to \mathbb{N}, x_i \mapsto n_i$, where $x_i \mapsto n_i$ states that the value of x_i under σ is n_i, or that n_i is stored in x_i, and is written as $\sigma(x_i) = n_i$.

We use the notation $\sigma[n/x]$ to represent a new state where the value of x is n and the values of the other variables are the same as in state σ. It is defined as

$$\sigma[n/x](y) = \begin{cases} n, & y = x, \\ \sigma(y), & y \neq x. \end{cases}$$

Refs. [21,22] propose that in a first-order language, a formal calculus including logical connectives and quantifiers can be built to deduce maximal consistent subsets of two inconsistent formal theories. When a failed test case of a program is found, it shows that the actual result of the program's execution is not consistent with the result expected from its specification. We can thus build a calculus to handle the inconsistency between "the actual execution" and "the expected execution"[2)]. To debug synchronous concurrent programs, this calculus is the structural operational semantics proposed in this paper.

The following notations are used to represent both the actual execution and the expected execution, and are important for semantics. $[\![e]\!]_\sigma$, $[\![b]\!]_\sigma$, and $[\![S]\!]_\sigma$ are used to represent the value of arithmetic expression e in σ, the truth value of boolean expression b in σ, and the state after executing statement S in σ, respectively. $[\![e]\!]_{\text{exp}}$, $[\![b]\!]_{\text{exp}}$, and $[\![S]\!]_{\text{exp}}$ are used to represent the expected value of e, the expected truth value of b, and the expected state after executing S, respectively.

Given the above concurrent programming language model and the notations, we will define the structural operational semantics for tracing and locating in the following two sections.

3 A structural operational semantics for tracing

The purpose of this section is to define a structural operational semantics for the tracing procedure. There are two points that should be taken into consideration. The first is the special role of assignment statements and communication statements, and the second relates to the nested branch structure[3)].

We first investigate the role of communication statements. When two processes communicate successfully, the receiving process assigns the received value, which is sent by the corresponding sending process, to a variable. Thus, the communication can be seen as a distributed assignment. This means that both

2) We assume that, for a given specification, there exists an expected program that will do exactly what the specification requires. The execution of such a program is the expected execution.

3) A special situation arises when the nesting depth is 1.

assignment statements and communication statements can change the values of variables and may cause an unexpected state.

The second issue is that **if** and **while** statements create nested branch structures. When a statement is being executed, it may be in some nested structure. When checking such statements, we should first check their nested structures. This is because if the nested structure is not as expected, the statement being checked becomes irrelevant. This means that when assignment statements and communication statements are recorded, the corresponding nested structures should also be recorded.

According to the characteristics of concurrent programs, we design the following data structures to record this information.

3.1 Global state σ

In a SyncCP, memory spaces for different processes are disjoint. Although a few variable names for different processes might be the same, they can still be uniquely identified, e.g., by using process id. For simplicity, we assume that variable names for different processes are different in a SyncCP. We can then use a global state σ to uniformly represent the mapping from the variables of a SyncCP to their values.

3.2 Global environment stack array ε and local environment stack $\varepsilon[pid]$ for process pid

Nested branch structures are generated by **if** and **while** statements and depend on the boolean expressions in these statements. We hence give the following two definitions:

Definition 2 (Environment)**.** Let b be a boolean expression that is the boolean condition of either an **if** statement or a **while** statement, and σ be a state in which b will be evaluated. The pair $\langle b, \sigma\rangle$ is called an environment.

Definition 3 (Environment stack)**.** A stack is called an environment stack if its elements are environments $\langle b, \sigma\rangle$.

An environment stack can be used to represent nested branch structure, i.e., the bottom element of an environment stack represents the outermost branch and the top element represents the innermost branch. Through σ, we can evaluate the truth value of b and check whether the truth value is as expected.

However, for a SyncCP, we cannot use a uniform global environment stack to record all the environments of different processes. Let us consider the example below.

Example 1 (Wrong environment stack for a SyncCP)**.** The left-hand side of Figure 2 shows a SyncCP consisting of two processes P_1 and P_2. P_1's statement **if** b **then** $\{S\}$ **else** $\{S'\}$ and P_2's statement **if** c **then** $\{A\}$ **else** $\{A'\}$ can be executed simultaneously. Suppose b of P_1 is evaluated first[4)] and c of P_2 is evaluated later. When using a global environment stack, environment $\langle b, \sigma_1\rangle$ corresponding to b will be pushed first into the stack, followed by environment $\langle c, \sigma_2\rangle$ corresponding to c. The resultant stack is shown on the right-hand side of Figure 2. Following this, if the execution of statement S or S' is completed first, the environment generated by **if** b **then** $\{S\}$ **else** $\{S'\}$ should be cleared in order to not affect subsequent recording. Our current strategy is simply to pop the top element of the stack, i.e., $\langle c, \sigma_2\rangle$. Therefore, when statement A or A' is executed, the corresponding recorded nested structure will be environment $\langle b, \sigma_1\rangle$. This is obviously wrong.

Thus, we design a local environment stack $\varepsilon[pid]$ for each process pid and design a global environment stack array ε that consists of these local environment stacks. We can then safely operate on $\varepsilon[pid]$ because it is local.

3.3 Global trace stack π

In order to record assignment statements and communication statements (i.e., sending statements and receiving statements), we provide the following definition:

4) Because the truth value of a boolean expression can influence the execution branch that follows, we treat the evaluation of the boolean expression, which leads to the corresponding environment being pushed into the environment stack, as a single execution step in this paper.

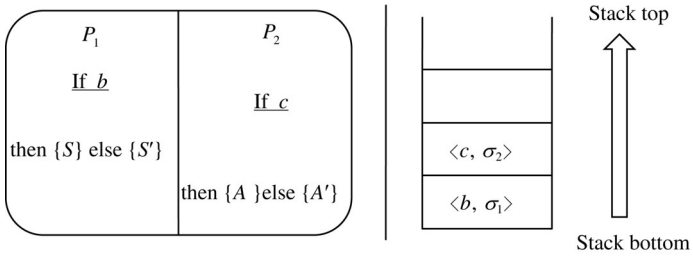

Figure 2 Wrong environment stack for a SyncCP.

Definition 4 (Local S-configuration)**.** Let σ be a state, S be an assignment, sending or receiving statement, of process *pid* to be executed in state σ, and ε be the environment stack array. The triple $\langle pid : S,\ \sigma,\ \varepsilon \rangle$ is called a local S-configuration.

The local S-configuration is used to record the scenario in which S of process *pid* was executed, including the statement $pid : S$, the state σ, and the environment stack array ε. It can be seen as the execution history of statement S.

By combining *pid* and ε, we can easily identify the local environment stack $\varepsilon[pid]$. Along with the assumption that variable names are different from each other in a SyncCP, we use a global stack to record these local S-configurations.

Definition 5 (Trace stack)**.** A stack π is called a trace stack if its elements are local S-configurations $\langle pid : S,\ \sigma,\ \varepsilon \rangle$.

The trace stack records the execution history of a program. Its elements are local S-configurations stored in a reverse order of program execution.

3.4 The structural operational semantics

The tracing procedure re-executes the concurrent program with the failed test case and uses the above data structures to record useful information, i.e., the execution history of the program. By using the above data structures, we shall define the structural operational semantics of tracing in this section. First, we give the following two definitions.

Definition 6 (Global S-configuration)**.** Let σ be a state, S be a statement of process *pid*, Q be the remaining processes excluding process *pid*, and ε be the environment stack array. The triple $\langle pid : S \parallel Q,\ \sigma,\ \varepsilon \rangle$ is called a global S-configuration.

Definition 7 (T-configuration)**.** Let $\langle pid : S \parallel Q,\ \sigma,\ \varepsilon \rangle$ be a global S-configuration, and π be a trace stack. A T-configuration is of the form $\pi \mid \langle pid : S \parallel Q,\ \sigma,\ \varepsilon \rangle$.

The global S-configuration shows the current execution status of the entire concurrent program, i.e., the current concurrent program $pid : S \parallel Q$, the current state σ, and the current environment stack array ε. Statement S of process *pid* will be executed in σ with ε. The trace stack π on the left side records local S-configurations whose statements have been executed before the global S-configuration. Thus, a T-configuration can be used to trace the execution of the program and to save the execution history for the locating procedure.

The structural operational semantics interprets the execution of a SyncCP by a sequence of computational steps, and each computational step is interpreted as a transition from the current T-configuration to the next T-configuration. The structural operational semantics consists of six groups of rules, where one group is related to communication and the others correspond to local statements[5)]. Each rule is a fraction, where the numerator represents the premises and the denominator represents the conclusion. The fraction means that if the conditions and transitions in the numerator hold, then the transition in the denominator is performed.

5) A statement of any process that is not a communication statement is called a local statement.

- Assignment statement

$$\frac{[\![e]\!]_\sigma = n}{\pi \mid \langle pid : x := e \mid\mid Q,\ \sigma,\ \varepsilon\rangle \longrightarrow push(\langle pid : x := e,\ \sigma,\ \varepsilon\rangle, \pi) \mid \langle pid : \mathbf{skip} \mid\mid Q,\ \sigma[n/x],\ \varepsilon\rangle}. \tag{3.1}$$

This rule states that the assignment statement of process *pid* is executed in state σ with environment stack array ε and trace stack π. If the value of expression e in state σ is n, then after the execution of one step, the assignment statement of process *pid* becomes **skip**, and other processes remain unchanged. The state is changed to $\sigma[n/x]$, the environment stack array is unchanged, and the local S-configuration $\langle pid : x := e,\ \sigma,\ \varepsilon\rangle$ is pushed into trace stack π.

- **if** statement

$$\frac{[\![b]\!]_\sigma = \text{True}}{\pi \mid \langle pid : \mathbf{if}\ b\ \mathbf{then}\ \{S_1\}\ \mathbf{else}\ \{S_2\} \mid\mid Q,\ \sigma,\ \varepsilon\rangle \longrightarrow \pi \mid \langle pid : \{S_1\} \mid\mid Q,\ \sigma,\ push(\langle b, \sigma\rangle, \varepsilon[pid])\rangle}, \tag{3.2}$$

$$\frac{[\![b]\!]_\sigma = \text{False}}{\pi \mid \langle pid : \mathbf{if}\ b\ \mathbf{then}\ \{S_1\}\ \mathbf{else}\ \{S_2\} \mid\mid Q,\ \sigma,\ \varepsilon\rangle \longrightarrow \pi \mid \langle pid : \{S_2\} \mid\mid Q,\ \sigma,\ push(\langle b, \sigma\rangle, \varepsilon[pid])\rangle}. \tag{3.3}$$

Rule (3.2) says that if the boolean expression b of **if** statement of process *pid* is evaluated to be true in state σ, the next statement to be executed in process *pid* is $\{S_1\}$. The other processes and state σ remain unchanged. $push(\langle b, \sigma\rangle, \varepsilon[pid])$ is used to push environment $\langle b, \sigma\rangle$ into local environment stack $\varepsilon[pid]$ and to leave other local environment stacks unchanged. The result is a new environment stack array. Rule (3.3) is similar to rule (3.2), except that if the boolean expression b is evaluated to be false in state σ, the next statement to be executed in process *pid* will be $\{S_2\}$.

- **while** statement

$$\frac{[\![b]\!]_\sigma = \text{True}}{\pi \mid \langle pid : \mathbf{while}\ b\ \mathbf{do}\ \{S\} \mid\mid Q,\ \sigma,\ \varepsilon\rangle \longrightarrow \pi \mid \langle pid : \{S\}; \mathbf{while}\ b\ \mathbf{do}\ \{S\} \mid\mid Q,\ \sigma,\ push(\langle b, \sigma\rangle, \varepsilon[pid])\rangle}, \tag{3.4}$$

$$\frac{[\![b]\!]_\sigma = \text{False}}{\pi \mid \langle pid : \mathbf{while}\ b\ \mathbf{do}\ \{S\} \mid\mid Q,\ \sigma,\ \varepsilon\rangle \longrightarrow \pi \mid \langle pid : \mathbf{skip} \mid\mid Q,\ \sigma,\ \varepsilon\rangle}. \tag{3.5}$$

Rule (3.4) is similar to rule (3.2), with the difference that the next statement to be executed in process *pid* is $\{S\}$; **while** b **do** $\{S\}$. This means that the body S of **while** statement is executed first, following which the **while** statement is executed again. Rule (3.5) says that if the boolean expression b of **while** statement of process *pid* is evaluated to be false in state σ, the **while** statement of process *pid* terminates. The other processes, the state σ and the environment stack array ε all remain unchanged.

- Communication

$$\frac{\pi \mid \langle Q,\ \sigma,\ \varepsilon\rangle \longrightarrow \pi' \mid \langle Q',\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle Q \mid\mid C!e,\ \sigma,\ \varepsilon\rangle \longrightarrow \pi' \mid \langle Q' \mid\mid C!e,\ \sigma',\ \varepsilon'\rangle}, \tag{3.6}$$

$$\frac{\pi \mid \langle Q,\ \sigma,\ \varepsilon\rangle \longrightarrow \pi' \mid \langle Q',\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle Q \mid\mid C?x,\ \sigma,\ \varepsilon\rangle \longrightarrow \pi' \mid \langle Q' \mid\mid C?x,\ \sigma',\ \varepsilon'\rangle}, \tag{3.7}$$

$$\frac{[\![e]\!]_\sigma = n}{\begin{array}{l}\pi|\langle pid_1 : C!e \mid\mid pid_2 : C?x \mid\mid Q,\ \sigma,\ \varepsilon\rangle \longrightarrow \\ push(\langle pid_1 : C!e,\ \sigma,\ \varepsilon\rangle,\ \langle pid_2 : C?x,\ \sigma,\ \varepsilon\rangle,\ \pi)|\langle pid_1 : \mathbf{skip} \mid\mid pid_2 : \mathbf{skip} \mid\mid Q,\ \sigma[n/x],\ \varepsilon\rangle\end{array}}. \tag{3.8}$$

The above three rules represent the semantics of synchronous communication, i.e., communication can be successful only when the sending process is ready to send and the receiving process is ready to receive. Otherwise, the process that has already reached the communication point should wait. Rules (3.6) and (3.7) represent situations when only one process reaches the communication point. Rule (3.6) says that if only the sending process reaches the communication point, it stays unchanged and other processes Q move forward if they can. Rule (3.7) is similar, but applies to the receiving process. These two rules are meant to preserve the integrity of the semantics and are not used during the tracing procedure because the function they perform can be carried out by other local statement rules. Rule (3.8) represents the situation when communication is successful. It says that communication can occur if process pid_1

reaches $C!e$ and process pid_2 reaches $C?x$. If the value of expression e in state σ is n, then following the execution of the communication, the next statements to be executed in processes pid_1 and pid_2 both become **skip**, the state σ is changed to $\sigma[n/x]$, the environment stack array ε is unchanged, and the local S-configurations $\langle pid_1 : C!e,\ \sigma,\ \varepsilon\rangle$ and $\langle pid_2 : C?x,\ \sigma,\ \varepsilon\rangle$ are pushed into the trace stack π. $push(A, B, \pi)$ pushes A into stack π followed by B.

- Sequential statement

$$\frac{\pi \mid \langle pid : S_1 \mid\mid Q,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi' \mid \langle pid : S_1{}' \mid\mid Q',\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle pid : S_1; S_2 \mid\mid Q,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi' \mid \langle pid : S_1{}'; S_2 \mid\mid Q',\ \sigma',\ \varepsilon'\rangle}, \tag{3.9}$$

$$\frac{\pi \mid \langle pid : S_1 \mid\mid Q,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi' \mid \langle pid : \mathbf{skip} \mid\mid Q',\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle pid : S_1; S_2 \mid\mid Q,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi' \mid \langle pid : S_2 \mid\mid Q',\ \sigma',\ \varepsilon'\rangle}. \tag{3.10}$$

Rule (3.9) states that for sequential statement $S_1; S_2$ of process pid, S_1 will be executed first. Assume that S_1 of process pid is executed in state σ with environment stack array ε, trace stack π, remaining processes Q. Following one execution step, S_1 becomes $S_1{}'$, σ is changed to σ', ε is changed to ε', π is changed to π', and Q becomes Q'. Then, following one execution step of the sequential statement $S_1; S_2$, the T-configuration becomes $\pi' \mid \langle pid : S_1{}'; S_2 || Q', \sigma', \varepsilon'\rangle$. Note that if S_1 contains communication statements, its execution can lead to the execution of some processes in Q, which changes Q to Q'. Rule (3.10) says that if, following one execution step of process pid, S_1 becomes **skip**, then $S_1; S_2$ will change to S_2. In other words, S_2 is the next statement to be executed in process pid if and only if S_1 terminates.

- Block

$$\pi \mid \langle pid : \{\mathbf{skip}\} \mid\mid Q,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi \mid \langle pid : \mathbf{skip} \mid\mid Q,\ \sigma,\ pop(\varepsilon[pid])\rangle, \tag{3.11}$$

$$\frac{\pi \mid \langle pid : S_1 \mid\mid Q,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi' \mid \langle pid : S_1{}' \mid\mid Q',\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle pid : \{S_1; S_2\} \mid\mid Q,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi' \mid \langle pid : \{S_1{}'; S_2\} \mid\mid Q',\ \sigma',\ \varepsilon'\rangle}, \tag{3.12}$$

$$\frac{\pi \mid \langle pid : S_1 \mid\mid Q,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi' \mid \langle pid : \mathbf{skip} \mid\mid Q',\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle pid : \{S_1; S_2\} \mid\mid Q,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi' \mid \langle pid : \{S_2\} \mid\mid Q',\ \sigma',\ \varepsilon'\rangle}. \tag{3.13}$$

The above three rules are used to specify the execution of a block in an environment $\langle b, \sigma_b\rangle$, which is the top element of local environment stack $\varepsilon[pid]$. The rules are similar to those of the sequential statement, except for rule (3.11). This rule states that when the statements belonging to the block of process pid terminate, the environment $\langle b, \sigma_b\rangle$ is popped out from $\varepsilon[pid]$. For the sake of convenience, we give the following communication $'$ rule (3.8$'$). It can be deduced from communication rule (3.8) and sequential statement rule (3.10). The proof is given in Appendix A.

- Communication$'$

$$\frac{[\![e]\!]_\sigma = n}{\begin{array}{l}\pi \mid \langle pid_1 : C!e; S_1 \mid\mid pid_2 : C?x; S_2 \mid\mid Q,\ \sigma,\ \varepsilon\rangle \longrightarrow \\ push(\langle pid_1 : C!e,\ \sigma,\ \varepsilon\rangle, \langle pid_2 : C?x,\ \sigma,\ \varepsilon\rangle, \pi) \mid \langle pid_1 : S_1 \mid\mid pid_2 : S_2 \mid\mid Q,\ \sigma[n/x],\ \varepsilon\rangle\end{array}}. \tag{3.8$'$}$$

Note that the semantics of synchronous communication guarantees that the execution order of local statements of different processes can be arbitrary. Thus, when several processes can be executed simultaneously, any one can be chosen first. In what follows, we give two simple examples to demonstrate the use of these rules.

Example 2. Figure 3(a) is a SyncCP used to exemplify the use of the communication rule. This SyncCP consists of two processes P_1 and P_2 communicating through channel C. P_2 uses the received value for some computation. Figure 3(b) shows the execution process and the trace stacks generated during this process. The step numbers indicate the concrete execution order. The statement(s) following the step number is (are) the statement(s) to be executed in the corresponding step. The stack above each step number is the trace stack generated after the execution of the step, and is represented by π_i. We now show how to get these trace stacks by using the rules of the semantics of tracing.

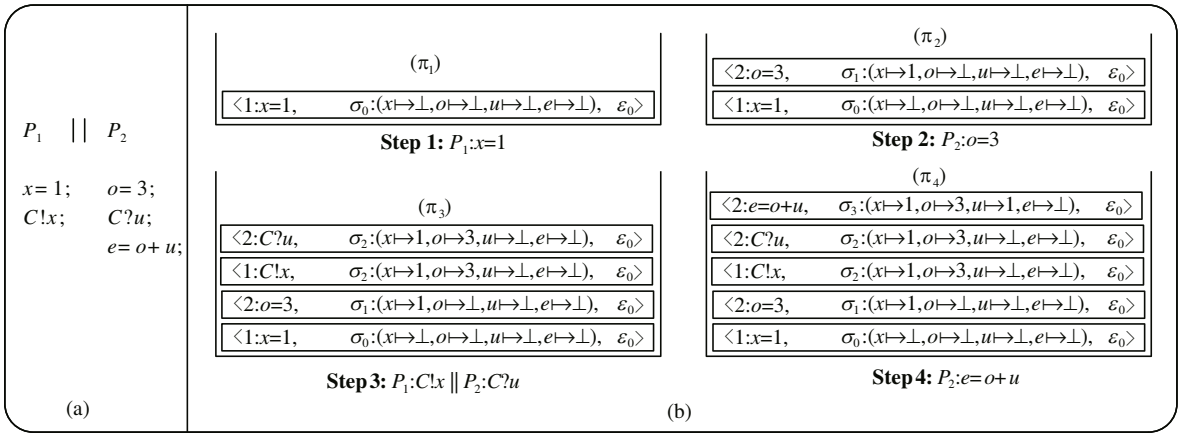

Figure 3 The tracing of a SyncCP. (a) The program; (b) the execution process and the trace stack.

In the initial T-configuration, the trace stack π_0 is empty ($\pi_0 = \emptyset$), the state σ_0 is $(x \mapsto \perp, o \mapsto \perp, u \mapsto \perp, e \mapsto \perp)$, and all the local environment stacks in the environment stack array ε_0 are empty ($\varepsilon_0 = [\emptyset, \emptyset]$), and will remain empty during execution because the program does not contain any **if** or **while** statements.

Step 1 The assignment statement of P_1 is chosen to be executed, and assignment statement rule (3.1) and sequential statement rule (3.10) are applied.

$$\frac{[\![1]\!]_{\sigma_0} = 1}{\pi_0 \mid \langle 1 : x := 1 \mid\mid Q,\ \sigma_0,\ \varepsilon_0\rangle \longrightarrow \underbrace{push(\langle 1 : x := 1,\ \sigma_0,\ \varepsilon_0\rangle, \pi_0)}_{\pi_1} \mid \langle 1 : \mathbf{skip} \mid\mid Q,\ \underbrace{\sigma_0[1/x]}_{\sigma_1},\ \varepsilon_0\rangle}\ (\text{by (3.1)})$$
$$\overline{\pi_0 \mid \langle 1 : x := 1; C!x \mid\mid Q,\ \sigma_0,\ \varepsilon_0\rangle \longrightarrow \pi_1 \mid \langle 1 : C!x \mid\mid Q,\ \sigma_1,\ \varepsilon_0\rangle}\ (\text{by (3.10)}).$$

Here, Q is P_2, which is $2 : o = 3; C?u; e = o + u$. Following this step, we get trace stack π_1, shown in Figure 3(b), and the new state $\sigma_1 = (x \mapsto 1, o \mapsto \perp, u \mapsto \perp, e \mapsto \perp)$. The current program becomes $1 : C!x \mid\mid 2 : o = 3; C?u; e = o + u$.

Step 2 Although process P_1 is ready to send, it should wait because the corresponding receiving statement is not ready. The first assignment statement of P_2 is chosen to be executed and, again, rules (3.1) and (3.10) are applied.

$$\frac{[\![3]\!]_{\sigma_1} = 3}{\pi_1 \mid \langle 2 : o := 3 \mid\mid \underbrace{1 : C!x}_{Q},\ \sigma_1,\ \varepsilon_0\rangle \longrightarrow \underbrace{push(\langle 2 : o := 3,\ \sigma_1,\ \varepsilon_0\rangle, \pi_1)}_{\pi_2} \mid \langle 2 : \mathbf{skip} \mid\mid \underbrace{1 : C!x}_{Q},\ \underbrace{\sigma_1[3/o]}_{\sigma_2},\ \varepsilon_0\rangle}\ (\text{by (3.1)})$$
$$\overline{\pi_1 \mid \langle 2 : o := 3; C?u; e = o + u \mid\mid \underbrace{1 : C!x}_{Q},\ \sigma_1,\ \varepsilon_0\rangle \longrightarrow \pi_2 \mid \langle 2 : C?u; e = o + u, 2 \mid\mid \underbrace{1 : C!x}_{Q},\ \sigma_2,\ \varepsilon_0\rangle}\ (\text{by (3.10)}).$$

Following this step, we get trace stack π_2, shown in Figure 3(b), and the new state $\sigma_2 = (x \mapsto 1, o \mapsto 3, u \mapsto \perp, e \mapsto \perp)$. The current program becomes $1 : C!x \mid\mid 2 : C?u; e = o + u$.

Step 3 P_1 is now ready to send and P_2 is ready to receive. The communication is thus chosen to be executed and rule (3.8′) is applied.

$$\frac{[\![x]\!]_{\sigma_2} = 1}{\pi_2 \mid \langle 1 : C!x \mid\mid 2 : C?u; e = o + u,\ \sigma_2,\ \varepsilon_0\rangle \longrightarrow \underbrace{push(\langle 1 : C!x,\ \sigma_2,\ \varepsilon_0\rangle, \langle 2 : C?u,\ \sigma_2,\ \varepsilon_0\rangle, \pi_2)}_{\pi_3} \mid \langle 1 : \mathbf{skip} \mid\mid 2 : e = o + u,\ \underbrace{\sigma_2[1/u]}_{\sigma_3},\ \varepsilon_0\rangle}\ (\text{by (3.8}')).$$

At this step, two local S-configurations corresponding to the communication statements are pushed into π_2. Following this step, we get trace stack π_3, shown in Figure 3(b), and the new state $\sigma_3 = (x \mapsto 1, o \mapsto 3, u \mapsto 1, e \mapsto \perp)$. The current program becomes $1 : \mathbf{skip} \mid\mid 2 : e = o + u$.

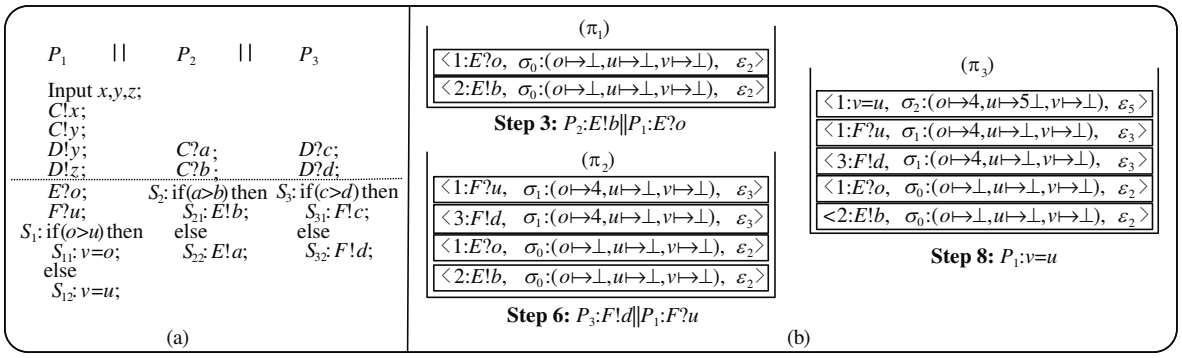

Figure 4 Get the maximum of three numbers. (a) The program; (b) the execution process and the trace stack.

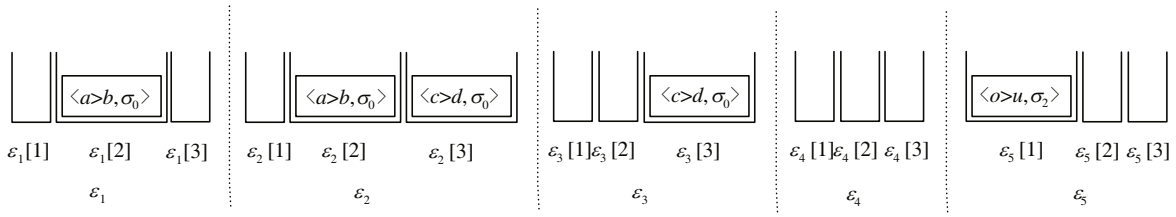

Figure 5 The environment stack arrays during the execution.

Step 4 Now P_1 terminates. The last assignment statement of P_2 is chosen to be executed and rule (3.1) is applied.

$$\frac{[\![o+u]\!]_{\sigma_3}=4}{\pi_3 \mid \langle 2: e=o+u \parallel \underbrace{1:\mathbf{skip}}_{Q},\ \sigma_3,\ \varepsilon_0\rangle \longrightarrow \underbrace{push(\langle 2: e=o+u,\ \sigma_3,\ \varepsilon_0\rangle, \pi_3)}_{\pi_4} \mid \langle 2:\mathbf{skip} \parallel \underbrace{1:\mathbf{skip}}_{Q},\ \underbrace{\sigma_3[4/e]}_{\sigma_4},\ \varepsilon_0\rangle} \text{ (by (3.1)).}$$

Following this step, we obtain the final trace stack π_4, shown in Figure 3(b), and the final state $\sigma_4 = (x \mapsto 1, o \mapsto 3, u \mapsto 1, e \mapsto 4)$.

Example 3 (Get the maximum of three numbers)**.** Figure 4(a) shows a SyncCP that aims to obtain the maximum of three numbers. The program consists of three processes P_1, P_2 and P_3. P_1 sends the first two numbers to P_2 and the last two numbers to P_3. P_2 and P_3 perform the same operation, i.e., to compute the maximum of the two received numbers and return it to P_1. Finally, in P_1, the maximum of the two returned numbers is computed, which is also the maximum of the three numbers. Although this SyncCP is simple, it demonstrates a scenario where a complicated task is divided into simple tasks that can be processed separately.

Assume that the input is $x = 6, y = 4, z = 5$ and that P_1 has already sent the numbers to P_2 and P_3. The statements to then be executed in each process are $E?o$, S_2, and S_3, respectively. The current trace stack π_0 contains the local S-configurations that are generated by the previous communications. The current state σ_0 is $(x \mapsto 6, y \mapsto 4, z \mapsto 5, a \mapsto 6, b \mapsto 4, c \mapsto 4, d \mapsto 5, o \mapsto \perp, u \mapsto \perp, v \mapsto \perp)$. The values of x, y, z, a, b, c, d are unchanged during the execution. For simplicity, we ignore the previous information. This means that we see π_0 as empty ($\pi_0 = \emptyset$) and σ_0 as $(o \mapsto \perp, u \mapsto \perp, v \mapsto \perp)$[6]. The initial environment stack array is ε_0, where each $\varepsilon_0[pid]$ is empty.

The content of Figure 4(b) is similar to that of Figure 3: it shows the execution process and the trace stacks generated during the process. The step number in Figure 4(b) is discontinuous because there are many steps that do not affect the trace stack, e.g., the step involving the use of the **if** statement rule. Figure 5 shows the change process of the environment stack array ε caused by **if** statements. For this example, we only provide a brief outline of the tracing process. The details can be found in Appendix B.

Note that the trace stacks mentioned in this paragraph can be found in Figure 4(b), and the environment stack arrays can be found in Figure 5. In Step 1, the **if** statement rule (3.2) is applied to statement S_2 of P_2, following which the environment stack array changes to ε_1. S_2 changes to $\{S_{21}\}$ because the

6) Actually, they contain previously collected information.

truth value of $a > b$ is true in σ_0. In Step 2, the **if** statement rule (3.2) is applied again to statement S_3 of P_3, following which the environment stack array changes to ε_2. S_3 changes to S_{32} because the truth value of $c > d$ is false in σ_0. In Step 3, the communication between statement $E?o$ of P_1 and statement S_{21} of P_2 is executed, through which we obtain trace stack π_1 and the new state $\sigma_1 = (o \mapsto 4, u \mapsto \perp, v \mapsto \perp)$. In Steps 4 and 5, block rules are applied to terminate the **if** statement S_2 of P_2, and the environment stack array changes to ε_3. In Step 6, using the same method as in Steps 3-5, for communication between $F?u$ of P_1 and S_{32} of P_3, we obtain the trace stack π_2 and the new state $\sigma_2 = (o \mapsto 4, u \mapsto 5, v \mapsto \perp)$. At the same time, the environment stack array changes to ε_4. Now both P_2 and P_3 terminate. In Step 7, the **if** statement rule (3.2) is applied to statement S_1 of P_1, following which the environment stack array changes to ε_5, and S_1 changes to $\{S_{12}\}$ because the truth value of $o > u$ is false in σ_2. In Step 8, assignment statement rule (3.1) is applied to statement S_{12}, following which we get the final trace stack π_3 and the final state $\sigma_3 = (o \mapsto 4, u \mapsto 5, v \mapsto 5)$. In the final step, the block rule is applied and the environment stack array changes to ε_6, which is the same as ε_0.

4 A structural operational semantics for locating

The locating procedure starts with the trace stack obtained in the tracing procedure. It aims to locate the ill-designed statement by backtracking the trace stack, and tries to fix the bug by solving the fix equation. In order to clearly describe this procedure, we first define the following concept, which is slightly different from the T-configuration proposed in Section 3.

Definition 8 (D-configuration)**.** The form $\pi \mid \langle pid : S, \sigma, \varepsilon \rangle$ is called a D-configuration.

D-configuration gives the configuration of a step during the locating procedure. π is the current trace stack, which is a sub-stack of the trace stack obtained in the tracing procedure. $\langle pid : S, \sigma, \varepsilon \rangle$ represents the current statement S, which is to be checked next. σ and ε are the state and the environment stack array, respectively, in which S was executed.

As discussed in Section 3, if a statement is in a nested structure, we should first check whether the nested structure is as expected. Only after confirming the nested structure should we proceed to check the statement; if the nested structure is not as expected, we should first investigate the reason for the unexpected nested structure. Based on this principle, we first present an outline of the locating procedure.

Step 1 The locating procedure starts immediately after the tracing procedure terminates and takes the following initial D-configuration $\pi \mid \langle x, \sigma, \varepsilon \rangle$[7)] as input. Because σ is the terminal state obtained by re-running the program with the failed test case, there exists at least one variable x whose value in σ is not as expected, i.e., $[\![x]\!]_\sigma \neq [\![x]\!]_{\exp}$.

Step 2 Pop the elements in π one by one until the first local S-configuration is found that contains a statement with an assignment effect on x. This operation can be implemented by a fas function such that $fas(x, \pi)$ is a sub-stack of π. The top element $\langle pid : S, \sigma', \varepsilon' \rangle$ of $fas(x, \pi)$ is the first local S-configuration from the top of π, which contains a statement that has an assignment effect on x. S can be an assignment statement or a receiving statement. Then go to Step 3 to check $\langle pid : S, \sigma', \varepsilon' \rangle$ with the new trace stack $pop(fas(x, \pi))$ which is a sub-stack of $fas(x, \pi)$ obtained by popping out the top element of $fas(x, \pi)$.

Step 3 For a given local S-configuration $\langle pid : S, \sigma', \varepsilon' \rangle$, check $\varepsilon'[pid]$. If $\varepsilon'[pid] = \emptyset$, statement S is either not in a nested structure or the nested structure is confirmed to be as expected, and we can directly check statement S according to the following three cases:

Step 3.1 If S is an assignment statement $pid : x := e$, the value of e in σ is not as expected; otherwise the value of x in σ is as expected. We have the following two cases.

Step 3.1.1 If there exists a variable $y \in FV(e)$, such that $[\![y]\!]_{\sigma'} \neq [\![y]\!]_{\exp}$. This means that the value of

7) Note that here π, σ and ε are the trace stack, the state, and the environment stack array, respectively, when the tracing procedure terminates. There is no process id for x because variable names are different from one another in a SyncCP.

y in state σ' is not as expected, i.e., the computation of y conducted prior to the current S-configuration is wrong. We thus return to Step 2 to check y, i.e., we backtrack to the first local S-configuration that contains a statement with an assignment effect on y.

Step 3.1.2 If for every $y \in FV(e)$, we have $[\![y]\!]_{\sigma'} = [\![y]\!]_{\exp}$. This means that the value of every free variable occurring in e is as expected, but the value of e is not as expected. There is only one possibility, i.e., the structure of arithmetic expression e is incorrect. For example, $x := y + z$ is incorrectly written as $x := y \cdot z$. Thus, an ill-designed statement is found and the locating procedure terminates. We denote the terminating D-configuration as $\pi \mid \langle Error(e), \sigma', \varepsilon' \rangle$. In this case, the locating procedure outputs the following fix equation $f(FV(P))_{\sigma'} = [\![e]\!]_{\exp}$. This equation is used to solve function f. An arithmetic expression can be obtained by applying f on $FV(P)$[8)], whose value in state σ' is the expected value of e. After solving the fix equation, we modify the assignment statement and re-execute the program with the failed test case.

Step 3.2 If S is a receiving statement $pid : C?x$, the received value is not as expected. Since the local environment stack has been confirmed to be as expected, it means that the communication has occurred in the expected place. Thus, the only reason for having received an unexpected value is that the value sent by the sending process is not as expected. We thus go back to Step 3 to check the corresponding local S-configuration $\langle pid' : C!e', \sigma'', \varepsilon'' \rangle$. This local S-configuration is the current top element of the trace stack because local S-configurations for communication statements are continuously pushed into the trace stack in the tracing procedure.

Step 3.3 If S is a sending statement $pid : C!e$, the value to be sent, i.e., the value of e, is not as expected. Similarly to Step 3.1, there are two cases.

Step 3.3.1 There exists a variable $y \in FV(e)$, such that $[\![y]\!]_{\sigma'} \neq [\![y]\!]_{\exp}$. This case is handled by the same method as used in Step 3.1.1.

Step 3.3.2 For every $y \in FV(e)$, we have $[\![y]\!]_{\sigma'} = [\![y]\!]_{\exp}$. This case is handled by the same method as used in Step 3.1.2.

Step 4 For a given local S-configuration $\langle pid : S, \sigma', \varepsilon' \rangle$, we check $\varepsilon'[pid]$. If $\varepsilon'[pid] \neq \emptyset$, we check each of the environments stored in $\varepsilon'[pid]$ starting from the top, i.e., from the innermost execution branch to the outermost. Suppose that the top element of $\varepsilon[pid]$ is $\langle b, \sigma_b \rangle$. There are two cases.

Step 4.1 $[\![b]\!]_{\sigma_b} = [\![b]\!]_{\exp}$. This means that the truth value of boolean expression b is as expected, i.e., the current execution branch is as expected. We simply pop out the top element and continue to check the next environment, i.e., the outer execution branch, if it exists. Thus, if $pop(\varepsilon'[pid])$ is empty, then go to Step 3), else go to Step 4) with the new environment stack $pop(\varepsilon'[pid])$.

Step 4.2 $[\![b]\!]_{\sigma_b} \neq [\![b]\!]_{\exp}$. This means that the truth value of boolean expression b is not as expected, i.e., the current execution branch is not as expected. There are two cases.

Step 4.2.1 There exists variable $y \in FV(b)$, such that $[\![y]\!]_{\sigma_b} \neq [\![y]\!]_{\exp}$. This means that the value of y in state σ_b is not as expected, i.e., the prior computation of the value of y is not as expected. Thus, we return to Step 2 and check the value of y, i.e., backtrack to the first local S-configuration that contains a statement with an assignment effect on y.

Step 4.2.2 For every $y \in FV(b)$, we have $[\![y]\!]_{\sigma_b} = [\![y]\!]_{\exp}$. This means that the value of every free variable occurring in b is as expected, but that the truth value of b is not as expected. There is only one possibility, i.e., the structure of boolean expression b is incorrect. For example, $x < y$ is incorrectly written as $x > y$. Thus, an ill-designed statement is found and the locating procedure terminates. The terminating D-configuration is $\pi \mid \langle Error(b), \sigma', \varepsilon' \rangle$. In this case, the locating procedure outputs the following fix equation $g(FV(P))_{\sigma_b} = [\![b]\!]_{\exp}$. This equation is used to solve function g. A boolean expression is obtained by applying g to $FV(P)$, whose truth value in state σ_b is the expected value of

8) $FV(P)$ represents the free variables that appear in the concurrent program. We use $FV(P)$ in addition to $FV(e)$ because the correct statement may contain variables that do not appear in e.

b. After solving the fix equation, we modify the boolean expression and re-execute the program with the failed test case.

According to the above outline, the structural operational semantics for the locating procedure is defined as follows:

- **Variables**

$$\frac{[\![x]\!]_\sigma \neq [\![x]\!]_{\exp},\quad top(fas(x,\pi)) = \langle pid : x := e, \sigma', \varepsilon'\rangle}{\pi \mid \langle x, \sigma, \varepsilon\rangle \ \longrightarrow\ pop(fas(x,\pi)) \mid \langle pid : x := e, \sigma', \varepsilon'\rangle}, \tag{4.1}$$

$$\frac{[\![x]\!]_\sigma \neq [\![x]\!]_{\exp},\quad top(fas(x,\pi)) = \langle pid : C?x, \sigma', \varepsilon'\rangle}{\pi \mid \langle x, \sigma, \varepsilon\rangle \ \longrightarrow\ pop(fas(x,\pi)) \mid \langle pid : C?x, \sigma', \varepsilon'\rangle}. \tag{4.2}$$

Rules (4.1) and (4.2) are the operational semantics for Step 2 in the outline. Taking rule (4.1) as an example, the inequality $[\![x]\!]_\sigma \neq [\![x]\!]_{\exp}$ in the numerator means that the value of x in σ is not as expected, which means that a bug has been encountered. The equality $top(fas(x,\pi)) = \langle pid : x := e, \sigma', \varepsilon'\rangle$ means that $\langle pid : x := e, \sigma', \varepsilon'\rangle$ is the first local S-configuration containing a statement that has an assignment effect on x, starting from the top of stack π. The operation in the denominator means that the next step in the locating procedure should check the statement which has an assignment effect on x.

- **Assignment statement**

$$\frac{\varepsilon[pid] = \emptyset,\ \ y \in FV(e),\ \ [\![y]\!]_\sigma \neq [\![y]\!]_{\exp},\ \ top(fas(y,\pi)) = \langle pid : y := e',\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle pid : x := e,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ pop(fas(y,\pi)) \mid \langle pid : y := e',\ \sigma',\ \varepsilon'\rangle}, \tag{4.3}$$

$$\frac{\varepsilon[pid] = \emptyset,\ \ y \in FV(e),\ \ [\![y]\!]_\sigma \neq [\![y]\!]_{\exp},\ \ top(fas(y,\pi)) = \langle pid : C?y,\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle pid : x := e,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ pop(fas(y,\pi)) \mid \langle pid : C?y,\ \sigma',\ \varepsilon'\rangle}, \tag{4.4}$$

$$\frac{\varepsilon[pid] = \emptyset,\ \ \forall y \in FV(e)([\![y]\!]_\sigma = [\![y]\!]_{\exp})}{\pi \mid \langle pid : x := e,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi \mid \langle Error(e),\ \sigma,\ \varepsilon\rangle}. \tag{4.5}$$

Rules (4.3) and (4.4) are the operational semantics for Step 3.1.1, and rule (4.5) is the operational semantics for Step 3.1.2.

- **Receiving statement**

$$\frac{\varepsilon[pid] = \emptyset,\ \ top(\pi) = \ \langle pid' : C!e, \sigma, \varepsilon\rangle}{\pi \mid \langle pid : C?x,\ \sigma,\ \varepsilon\rangle \longrightarrow pop(\pi) \mid \langle pid' : C!e, \sigma, \varepsilon\rangle}. \tag{4.6}$$

Rule (4.6) is the operational semantics for Step 3.2.

- **Sending statement**

$$\frac{\varepsilon[pid] = \emptyset,\ \ y \in FV(e),\ \ [\![y]\!]_\sigma \neq [\![y]\!]_{\exp},\ \ top(fas(y,\pi)) = \langle pid : y := e',\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle pid : C!e,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ pop(fas(y,\pi)) \mid \langle pid : y := e',\ \sigma',\ \varepsilon'\rangle}, \tag{4.7}$$

$$\frac{\varepsilon[pid] = \emptyset,\ \ y \in FV(e),\ \ [\![y]\!]_\sigma \neq [\![y]\!]_{\exp},\ \ top(fas(y,\pi)) = \langle pid : D?y,\ \sigma',\ \varepsilon'\rangle}{\pi \mid \langle pid : C!e,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ pop(fas(y,\pi)) \mid \langle pid : D?y,\ \sigma',\ \varepsilon'\rangle}, \tag{4.8}$$

$$\frac{\varepsilon[pid] = \emptyset,\ \ \forall y \in FV(e)([\![y]\!]_\sigma = [\![y]\!]_{\exp})}{\pi \mid \langle pid : C!e,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi \mid \langle Error(e),\ \sigma,\ \varepsilon\rangle}. \tag{4.9}$$

Rules (4.7) and (4.8) are the operational semantics for Step 3.3.1 and Rule (4.9) is the operational semantics for Step 3.3.2. These rules are very similar to rules for assignment statement, except that arithmetic expression e in them is part of the sending statement.

- **Block**

$$\frac{top(\varepsilon[pid]) = \langle b, \sigma_b\rangle,\ \ [\![b]\!]_{\sigma_b} = [\![b]\!]_{\exp}}{\pi \mid \langle pid : S,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ \pi \mid \langle pid : S,\ \sigma,\ \varepsilon[\varepsilon[pid] = pop(\varepsilon[pid])]\rangle}. \tag{4.10}$$

Rule (4.10) is the operational semantics of Step 4.1 in the outline. $\varepsilon[\varepsilon[pid] = pop(\varepsilon[pid])]$ is the new environment stack array obtained by popping out the top element of $\varepsilon[pid]$ and leaving the remaining local environment stacks unchanged.

$$\frac{\begin{pmatrix} top(\varepsilon[pid]) = \langle b, \sigma_b\rangle,\ \ [\![b]\!]_{\sigma_b} \neq [\![b]\!]_{\exp} \\ y \in FV(b),\ \ [\![y]\!]_{\sigma_b} \neq [\![y]\!]_{\exp},\ \ top(fas(y,\pi)) = \langle pid : y := e',\ \sigma',\ \varepsilon'\rangle \end{pmatrix}}{\pi \mid \langle pid : S,\ \sigma,\ \varepsilon\rangle \ \longrightarrow\ pop(fas(y,\pi)) \mid \langle pid : y := e',\ \sigma',\ \varepsilon'\rangle}, \tag{4.11}$$

$$\frac{\begin{pmatrix} top(\varepsilon[pid]) = \langle b, \sigma_b \rangle, \quad [\![b]\!]_{\sigma_b} \neq [\![b]\!]_{\exp} \\ y \in FV(b), \quad [\![y]\!]_{\sigma_b} \neq [\![y]\!]_{\exp}, \quad top(fas(y,\pi)) = \langle pid : C?y,\ \sigma',\ \varepsilon' \rangle \end{pmatrix}}{\pi \mid \langle pid : S,\ \sigma,\ \varepsilon \rangle \ \longrightarrow\ pop(fas(y,\pi)) \mid \langle pid : C?y,\ \sigma',\ \varepsilon' \rangle}. \tag{4.12}$$

Rules (4.11) and (4.12) are the operational semantics for Step 4.2.1 in the outline.

$$\frac{top(\varepsilon[pid]) = \langle b, \sigma_b \rangle, \quad [\![b]\!]_{\sigma_b} \neq [\![b]\!]_{\exp}, \quad \forall y \in FV(b)([\![y]\!]_{\sigma_b} = [\![y]\!]_{\exp})}{\pi \mid \langle pid : S,\ \sigma,\ \varepsilon \rangle \ \longrightarrow\ \pi \mid \langle Error(b),\ \sigma,\ \varepsilon \rangle}. \tag{4.13}$$

Rule (4.13) is the operational semantics for Step 4.2.2 in the outline.

To demonstrate the use of the operational semantics of locating, let us consider Example 2 and Example 3 from Subsection 3.4 again.

Example 4. In Example 2, the expected result is $e = 5$, but the result obtained by executing the failed test case is $e = 4$. The initial D-configuration is $\pi_4 | \langle e,\ \sigma_4,\ \varepsilon \rangle$, where ε is $[\emptyset, \emptyset]$. The process of locating the error by using the operational semantics of locating (or the locating procedure) is as follows:

(a) $$\frac{[\![e]\!]_{\sigma_4} \neq [\![e]\!]_{\exp}, \quad top(fas(e,\pi_4)) = \langle 2 : e := o + u,\ \sigma_3,\ \varepsilon \rangle}{\pi_4 \mid \langle e,\ \sigma_4,\ \varepsilon \rangle \longrightarrow \underbrace{pop(fas(e,\pi_4))}_{\pi_3} \mid \langle 2 : e := o + u,\ \sigma_3,\ \varepsilon \rangle}$$ (by (4.1)),

(b) $$\frac{\varepsilon[2] = \emptyset, \quad [\![u]\!]_{\sigma_3} \neq [\![u]\!]_{\exp}, \quad top(fas(u,\pi_3)) = \langle 2 : C?u,\ \sigma_2,\ \varepsilon \rangle}{\pi_3 \mid \langle 2 : e := o + u,\ \sigma_3,\ \varepsilon \rangle \longrightarrow \underbrace{pop(fas(u,\pi_3))}_{\pi'_3} \mid \langle 2 : C?u,\ \sigma_2,\ \varepsilon \rangle}$$ (by (4.4)),

(c) $$\frac{\varepsilon[2] = \emptyset, \quad top(\pi'_3) = \langle 1 : C!x,\ \sigma_2,\ \varepsilon \rangle}{\pi'_3 \mid \langle 2 : C?u,\ \sigma_2,\ \varepsilon \rangle \longrightarrow \underbrace{pop(\pi'_3)}_{\pi_2} \mid \langle 1 : C!x,\ \sigma_2,\ \varepsilon \rangle}$$ (by (4.6)),

(d) $$\frac{[\![C!x]\!]_{\sigma_2} \neq [\![C!x]\!]_{\exp}, \quad [\![x]\!]_{\sigma_2} = [\![x]\!]_{\exp}}{\pi_2 \mid \langle 1 : C!x,\ \sigma_2,\ \varepsilon \rangle \longrightarrow \pi_2 \mid \langle Error(x),\ \sigma_2,\ \varepsilon \rangle}$$ (by (4.8)).

Note that, in Step (b) above, $pop(fas(u,\pi_3))$ is denoted as π'_3, which is not π_2. This is because $fas(u,\pi_3)$ is obtained by pushing $\langle 1 : C!x,\ \sigma_2,\ \varepsilon \rangle$ and $\langle 2 : C?u,\ \sigma_2,\ \varepsilon \rangle$ together into π_2. In Step (d), fix equation $2 = (f(FV(P)))_{\sigma_2}$ is generated. Since the expected value of variable x is 1 and the expected value of the arithmetic expression in the sending statement is 2, a solution of the fix equation is $f(FV(P)) = x + 1$. Thus, one possible fix to the program is changing the sending statement $C!x$ into $C!(x+1)$, which can be confirmed by re-executing the program with the failed test case.

Example 5. Consider the program in Example 3. Suppose that the input is $x = 6, y = 4, z = 5$. The expected final result should be $v = 6$ rather than $v = 5$. The initial D-configuration is $\pi_3 \mid \langle v,\ \sigma_3,\ \varepsilon_6 \rangle$. The process of locating the error by using the operational semantics of locating (or the locating procedure) is as follows:

(a) $$\frac{[\![v]\!]_{\sigma_3} \neq [\![v]\!]_{\exp}, \quad top(fas(v,\pi_3)) = \langle 1 : v := u,\ \sigma_2,\ \varepsilon_5 \rangle}{\pi_3 \mid \langle v, \sigma_3, \varepsilon_6 \rangle \longrightarrow \underbrace{pop(fas(v,\pi_3))}_{\pi_2} \mid \langle 1 : v := u,\ \sigma_2,\ \varepsilon_5 \rangle}$$ (by (4.1)),

(b) $$\frac{\begin{pmatrix} top(\varepsilon_5[1]) = \langle o > u, \sigma_2 \rangle, \quad [\![o > u]\!]_{\sigma_2} \neq [\![o > u]\!]_{\exp} \\ [\![o]\!]_{\sigma_2} \neq [\![o]\!]_{\exp}, \quad top(fas(o,\pi_2)) = \langle 1 : E?o,\ \sigma_0,\ \varepsilon_2 \rangle \end{pmatrix}}{\pi_2 \mid \langle 1 : v := u,\ \sigma_2,\ \varepsilon_5 \rangle \longrightarrow \underbrace{pop(fas(o,\pi_2))}_{\pi'_2} \mid \langle 1 : E?o,\ \sigma_0,\ \varepsilon_2 \rangle}$$ (by (4.12)),

(c) $$\frac{\varepsilon_2[1] = \emptyset, \quad top(\pi'_2) = \langle 2 : E!b,\ \sigma_0,\ \varepsilon_2 \rangle}{\pi'_2 \mid \langle 1 : E?o,\ \sigma_0,\ \varepsilon_2 \rangle \longrightarrow \underbrace{pop(\pi'_2)}_{\pi_1} \mid \langle 2 : E!b,\ \sigma_0,\ \varepsilon_2 \rangle}$$ (by (4.6)),

(d) $$\frac{top(\varepsilon_2[2]) = \langle a > b \rangle, \quad [\![a > b]\!]_{\sigma_0} = [\![a > b]\!]_{\exp}}{\pi_1 \mid \langle 2 : E!b,\ \sigma_0,\ \varepsilon_2 \rangle \longrightarrow \pi_1 \mid \langle 2 : E!b,\ \sigma_0,\ \varepsilon[\varepsilon[2] = pop(\varepsilon_2[2])] \rangle}$$ (by (4.10)),

(e) $$\frac{\varepsilon[2] = \emptyset, \quad [\![E!b]\!]_{\sigma_0} \neq [\![E!b]\!]_{\exp}, \quad [\![b]\!]_{\sigma_0} = [\![b]\!]_{\exp}}{\pi_1 \mid \langle 2 : E!b,\ \sigma_0,\ \varepsilon \rangle \longrightarrow \pi_1 \mid \langle Error(b),\ \sigma_0,\ \varepsilon \rangle}$$ (by (4.5)).

As in Example 4, $pop(fas(o, \pi_2))$ is denoted as π_2'. In Step (e), fix equation $6 = g(FV(P))_{\sigma_0}$ is generated. Since the expected value of the arithmetic expression in the sending statement is 6, which is the value of variable a, a solution to the fix equation is $g(FV(P)) = a$. Thus, one possible fix to the program is to change the statement $E!b$ to $E!a$, which can be confirmed by re-executing the program with the failed test case.

5 Related work

Many debugging techniques have been proposed for sequential programs, such as step breakpoint [23], omniscient debugging [24], program slice [25], scientific debugging [2] and delta debugging [26].

For message passing-based concurrent programs, many mathematical models, as well as testing and debugging methods have also been proposed.

Mathematical models can be used to describe the action of concurrent programs. Petri net [3] is used to represent distributed concurrent system. A Petri net is a directed graph where the nodes represent transitions (i.e., events that may occur) and places (i.e., conditions). The directed arcs designate which places (conditions) are preconditions and/or postconditions for which transitions (events). The dependencies among events can represent the interactions among processes. Actor model [4] describes the character of message passing more directly. In actor model, everything is an actor. Different actors can communicate only through message passing. Another category of models is process calculus, which includes the Calculus of Communicating System (CCS) [5], Communicating Sequential Process (CSP) [6], Hybrid CSP (HCSP) [7], and so on. Process calculus describes the interaction, communication and synchronization among processes through algebraic approaches. All models have rigorous mathematical foundations and can be further used for program verification [8].

Structural testing [9] is based on the control structures of programs and has been extended to test message passing-based concurrent programs. This kind of methods [10–12] design different testing criteria to guide test case generation or to evaluate the quality of test cases. Reachability testing [13] is another testing method that focuses on synchronization (SYN)-sequences [14] of concurrent programs. Given an input, a concurrent program can have many SYN sequences, some of which may contain the error. The target of reachability testing is to traverse all possible SYN-sequences of a concurrent program to check for the error.

Debugging techniques for message passing-based concurrent programs collect the execution information of the programs and aid the debugging process using this information. One direction of investigation is to replay the previous execution by using the recorded execution events [15,16]. These approaches are orthogonal to our method and can aid our locating procedure. The other direction is to directly use the information to help analyze the behavior of the program. Xab [17] helps debug PVM programs by monitoring their runtime information and providing direct feedback about the functions being executed. Umpire [18] records programs' MPI operations by interposing itself between the programs and the MPI runtime system and then checks the programs' behavior for specific errors. Although these techniques expect to utilize the collected information, they do not propose a uniform method to locate the bug. Instead, our method provides a uniform framework to locate bugs for different programming languages and for different situations. This will greatly improve debugging efficiency .

Ref. [20] has proposed a debugging framework for sequential programs built up from notations and techniques of operational semantics. Our method is inspired by this method, but we extend it to synchronous message passing-based concurrent programs.

6 Conclusion

In this paper, we proposed a formal semantics for debugging synchronous message passing-based concurrent programs. We presented three programs written in different programming languages, and designed a concurrent programming language model based on these languages. Compared with sequential program-

ming languages, our model contains communication statements and concurrent structure to represent communication and concurrency. Our proposed debugging process consists of a tracing procedure and a locating procedure. The tracing procedure aims to collect useful information by re-executing the program with the failed test case. During this process, data structures, such as state, environment stack array and trace stack, are designed to record this information. A structural operational semantics, which represents the semantics of synchronous communication and concurrency, is provided to instruct the execution of tracing procedure. The locating procedure uses the information obtained in the tracing procedure to locate the ill-designed statement, generates a fix equation and tries to fix the bug by solving the fix equation. We also supply the structural operational semantics to instruct the locating procedure.

Ref. [20] has proposed a formal framework for debugging sequential programs. This paper improves this result by enabling the debugging of synchronous message passing-based concurrent programs. We have in mind at least three directions for future research. The first is to develop a prototype system based on our method and to use practical cases to evaluate the effectiveness of our method. The second direction is to design the formal framework for two other kinds of concurrent programs, i.e., asynchronous message passing-based concurrent programs and shared variable-based concurrent programs. The third direction in which we plan to pursue research is to relax the limitation whereby the expected value of each variable or expression has to be known, and to locate the error by using incomplete information.

Acknowledgements

This work was supported by State Key Laboratory of Software Development Environment (Grant No. SKLSDE-2012ZX-18).

References

1 Hailpern B, Santhanam P. Software debugging, testing and verification. IBM Syst J, 2002, 41: 4–12
2 Zeller A. Why Programs Fail. 2nd ed. Oxford: Elsevier Inc., 2009. 10–15
3 Petri C A, Reisig W. Petri net. Scholarpedia, 2008, 3: 64–77
4 Hewitt C, Bishop P, Steiger R. A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, San Francisco, 1973. 235–245
5 Milner R. A Calculus of Communicating Systems. Berlin: Springer Verlag, 1980. 65–83
6 Hoare C A R. Communicating sequential processes. Commun ACM, 1978, 21: 666–677
7 Zhou C C, Wang J, Anders P R. A formal description of hybrid systems. Hybrid Syst, 1996, 1066: 511–530
8 Hennessy M, Lin H M. Proof systems for message-passing process algebras. Form Asp Comput, 1996, 8: 379–407
9 Mrinal N D K C, Jamatia A. A review and analysis of software complexity metrics in structural testing. Int J Comput Commun Eng, 2013, 2: 129–133
10 Souza S R S, Vergilio S R, Souza P S L, et al. Structural testing criteria for message passing parallel programs. Concurr Comput Pract Exp, 2008, 20: 1893–1916
11 Taylor R N, Levine D L, Kelly C. Structural testing of concurrent programs. IEEE Trans Softw Eng, 1992, 18: 206–215
12 Yang R D, Chung C G. Path analysis testing of concurrent programs. Inf Softw Technol, 1992, 34: 43–56
13 Yu L, Carver R H. Reachability testing of concurrent programs. IEEE Trans Softw Eng, 2006, 32: 382–403
14 Carver R H, Tai K C. Replay and testing for concurrent programs. IEEE Softw, 1991, 8: 66–74
15 Leblanc T J, Mellor J M. Debugging parallel programs with instant replay. IEEE Trans Comput, 1987, 36: 471–482
16 Netzer R H B, Miller B P. Optimal tracing and replay for debugging message-passing parallel programs. In: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, Los Alamitos, 1992. 502–511
17 Beguelin A L. XAB: a tool for monitoring PVM programs. In: Proceedings of Workshop on Heterogeneous Processing. New York: IEEE, 1993. 92–97
18 Vetter J S, Supinski B R. Dynamic software testing of MPI applications with Umpire. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing. Piscataway: IEEE, 2000. 1–10
19 Plotkin G D. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19. 1981
20 Li W, Li N. A formal semantics for program debugging. Sci China Inf Sci, 2012, 55: 133–148
21 Li W. R-calculus: an inference system for belief revision. Comput J, 2007, 50: 378–390
22 Li W. Mathematical Logic: Foundations for Information Science. Basel: Birkhäuser Publisher, 2010. 173–192
23 Zhang Y K. Software Debugging. Beijing: Publishing House of Electronics Industry, 2009. 93–110
24 Lewis B. Debugging backwards in time. In: Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging, Ghent, 2003. 1–15

25 Weiser M. Program slicing. IEEE Trans Softw Eng, 1984, 10: 352–357
26 Misherghi G, Su Z. HDD: hierarchical delta debugging. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, 2006. 142–151

Appendix A Communication′ rule

Communication′ rule is described as (3.8′). For convenience, we usually use communication′ rule (3.8′) to replace communication rule (3.8). It can be deduced by communication rule (3.8) and sequential statement rule (3.10). The process is as follows.

- The first step is to use communication rule (3.8).

$$\frac{[\![e]\!]_\sigma = n}{\pi \mid \langle pid_1 : C!e \mid\mid pid_2 : C?x \mid\mid Q,\ \sigma,\ \varepsilon\rangle \longrightarrow \underbrace{push(\langle pid_1 : C!e,\ \sigma,\ \varepsilon\rangle, \langle pid_2 : C?x,\ \sigma,\ \varepsilon\rangle, \pi)}_{\pi'} \mid \langle pid_1 : \mathbf{skip} \mid\mid pid_2 : \mathbf{skip} \mid\mid Q,\ \sigma[n/x],\ \varepsilon\rangle} \text{ (by (3.8)).}$$

- The second step is to use sequential statement rule (3.10), where the handled process is the sending process.

$$\frac{\pi \mid \langle pid_1 : C!e \mid\mid \underbrace{pid_2 : C?x \mid\mid Q}_{Q_1},\ \sigma,\ \varepsilon\rangle \longrightarrow\ \pi' \mid \langle pid_1 : \mathbf{skip} \mid\mid \underbrace{pid_2 : \mathbf{skip} \mid\mid Q}_{Q_1'},\ \sigma[n/x],\ \varepsilon\rangle}{\pi \mid \langle pid_1 : C!e; S_1 \mid\mid \underbrace{pid_2 : C?x \mid\mid Q}_{Q_1},\ \sigma,\ \varepsilon\rangle \longrightarrow \pi' \mid \langle pid_1 : S_1 \mid\mid \underbrace{pid_2 : \mathbf{skip} \mid\mid Q}_{Q_1'},\ \sigma[n/x],\ \varepsilon\rangle} \text{ (by (3.10)).}$$

- The third step is to use sequential statement rule (3.10) again, where the handled process now is the receiving process and the initial T-configuration is the final T-configuration of step 2.

$$\frac{\pi \mid \langle pid_2 : C?x \mid\mid \underbrace{pid_1 : C!e; S_1 \mid\mid Q}_{Q_2},\ \sigma,\ \varepsilon\rangle \longrightarrow \pi' \mid \langle pid_2 : \mathbf{skip} \mid\mid \underbrace{pid_1 : S_1 \mid\mid Q}_{Q_2'},\ \sigma[n/x],\ \varepsilon\rangle}{\pi \mid \langle pid_2 : C?x; S_2 \mid\mid \underbrace{pid_1 : C!e; S_1 \mid\mid Q}_{Q_2},\ \sigma,\ \varepsilon\rangle \longrightarrow \pi' \mid \langle pid_2 : S_2 \mid\mid \underbrace{pid_1 : S_1 \mid\mid Q}_{Q_2'},\ \sigma[n/x],\ \varepsilon\rangle} \text{ (by (3.10)).}$$

The denominator is just the denominator of the communication′ rule.

Appendix B Detailed execution process of Example 3

In the following, we show how to use rules of the semantics of tracing for Example 3. Note that the trace stacks mentioned in this section can be found in Figure 4(b) and the environment stack arrays can be found in Figure 5.

Step 1 The **if** statement S_2 of P_2 is chosen to be executed and the **if** statement rule (3.2) is applied.

$$\frac{[\![a > b]\!]_{\sigma_0} = \text{True}}{\pi_0 \mid \langle 2 : \mathbf{if}\ a > b\ \mathbf{then}\ \{S_{21}\}\ \mathbf{else}\ \{S_{22}\} \mid\mid Q,\ \sigma_0,\ \varepsilon_0\rangle \longrightarrow \pi_0 \mid \langle 2 : \{S_{21}\} \mid\mid Q,\ \sigma_0,\ \underbrace{push(\langle a > b, \sigma_0\rangle, \varepsilon_0[2])}_{\varepsilon_1}\rangle} \text{ (by (3.2)).}$$

Here

$$Q := \underbrace{1 : E?o; F?u; S_1}_{P_1} \mid\mid \underbrace{3 : S_3}_{P_3}.$$

After this step, the environment stack array changes to ε_1 and S_2 becomes $\{S_{21}\}$. The current program becomes $1 : E?o; F?u; S_1 \mid\mid 2 : \{S_{21}\} \mid\mid 3 : S_3$.

Step 2 The **if** statement S_3 of P_3 is chosen to be executed and the **if** statement rule (3.2) is applied again.

$$\frac{[\![c > d]\!]_{\sigma_0} = \text{False}}{\pi_0 \mid \langle 3 : \mathbf{if}\ c > d\ \mathbf{then}\ \{S_{31}\}\ \mathbf{else}\ \{S_{32}\} \mid\mid Q,\ \sigma_0,\ \varepsilon_1\rangle \longrightarrow \pi_0 \mid 3 : \{S_{32}\} \mid\mid Q,\ \sigma_0,\ \underbrace{push(\langle c > d, \sigma_0\rangle, \varepsilon_1[3])}_{\varepsilon_2}\rangle} \text{ (by (3.2)).}$$

Here

$$Q := \underbrace{1 : E?o; F?u; S_1}_{P_1} \mid\mid \underbrace{3 : \{S_{21}\}}_{P_2}.$$

After this step, the environment stack array changes to ε_2 and S_3 becomes $\{S_{32}\}$. The current program becomes $1 : E?o; F?u; S_1 \mid\mid 2 : \{S_{21}\} \mid\mid 3 : \{S_{32}\}$.

Step 3 The communication between statement $E?o$ of P_1 and statement S_{21} of P_2 is chosen to be executed and the communication$'$ rule (3.8$'$) is applied. After the execution of the communication, block rule (3.13) is further applied.

$$\frac{\dfrac{[\![b]\!]_{\sigma_0}=4}{\pi_0 \mid \langle 2:E!b \parallel 1:E?o;F?u;S_1 \parallel \underbrace{3:\{S_{32}\}}_{Q},\ \sigma_0,\ \varepsilon_2\rangle \longrightarrow} \text{(by (3.8}')\text{)}}{\underbrace{push(\langle 2:E!b,\ \sigma_0,\ \varepsilon_2\rangle,\langle 1:E?o,\ \sigma_0,\ \varepsilon_2\rangle,\pi_0)}_{\pi_1} \mid \langle 2:\mathbf{skip} \parallel 1:F?u;S_1 \parallel \underbrace{3:\{S_{32}\}}_{Q},\ \underbrace{\sigma_0[4/o]}_{\sigma_1},\ \varepsilon_2\rangle} \text{(by (3.13))}.$$

When using rule (3.13), Q_1 is $\underbrace{1:E?o;F?u;S_1}_{P_1} \parallel \underbrace{3:\{S_{32}\}}_{P_3}$ and Q_1' is $\underbrace{1:F?u;S_1}_{P_1} \parallel \underbrace{3:\{S_{32}\}}_{P_3}$. At this step, two local S-configurations corresponding to the communication statements are pushed to π_0. After this step, we get trace stack π_1 and the new state $\sigma_1=(o\mapsto 4,u\mapsto\perp,v\mapsto\perp)$. The current program becomes $1:F?u;S_1 \parallel 2:\{\mathbf{skip};\} \parallel 3:\{S_{32}\}$.

Step 4 For the result of step 3, block rule (3.11) is applied for the termination of the **if** statement S_2 of P_2.

$$\pi_1 \mid \langle 2:\{\mathbf{skip}\} \parallel Q,\ \sigma_1,\ \varepsilon_2\rangle \longrightarrow \pi_1 \mid \langle 2:\mathbf{skip} \parallel Q,\ \sigma_1,\ \underbrace{pop(\varepsilon_2[2])}_{\varepsilon_3}\rangle \text{ (by (3.11))}.$$

Here

$$Q:=\underbrace{1:F?u;S_1}_{P_1} \parallel \underbrace{3:\{S_{32}\}}_{P_3}.$$

After this step, the environment stack array changes to ε_3. The current program becomes $1:F?u;S_1 \parallel 2:\mathbf{skip} \parallel 3:\{S_{32}\}$.

Step 5 According to step 3 and step 4, we can get

$$\pi_0 \mid \langle 2:\{S_{21}\} \parallel 1:E?o;F?u;S_1 \parallel 3:\{S_{32}\},\ \sigma_0,\ \varepsilon_2\rangle \longrightarrow \pi_1 \mid \langle 2:\mathbf{skip} \parallel 1:F?u;S_1 \parallel 3:\{S_{32}\},\ \sigma_1,\ \varepsilon_3\rangle.$$

Step 6 Similar to the above three steps, for the communication between $F?u$ of P_1 and S_{32} of P_3, we can get

$$\pi_1 \mid \langle 3:\{S_{32}\} \parallel 1:F?u;S_1 \parallel 2:\mathbf{skip},\ \sigma_1,\ \varepsilon_3\rangle \longrightarrow \pi_2 \mid \langle 3:\mathbf{skip} \parallel 1:S_1 \parallel 2:\mathbf{skip},\ \sigma_2,\ \varepsilon_4\rangle.$$

After this step, we can get the new trace stack $\pi_2=push(\langle 3:F!d,\ \sigma_1,\ \varepsilon_3\rangle,\langle 1:F?u,\ \sigma_1,\ \varepsilon_3\rangle,\pi_1)$, the new state $\sigma_2=\sigma_1[5/u]=(o\mapsto 4,u\mapsto 5,v\mapsto\perp)$, and the new environment stack array $\varepsilon_4=pop(\varepsilon_3[3])$. The current program becomes $1:S_1 \parallel 2:\mathbf{skip} \parallel 3:\mathbf{skip}$.

Step 7 Now processes P_2 and P_3 are terminated. The **if** statement S_1 of P_1 is chosen to be executed and the **if** statement rule (3.2) is applied.

$$\frac{[\![o>u]\!]_{\sigma_2}=False}{\pi_2 \mid \langle 1:\mathbf{if}\ o>u\ \mathbf{then}\ \{S_{11}\}\ \mathbf{else}\ \{S_{12}\} \parallel Q,\ \sigma_2,\ \varepsilon_4\rangle \longrightarrow \pi_2 \mid 1:\{S_{12}\} \parallel Q,\ \sigma_2,\ \underbrace{push(\langle o>u,\sigma_2\rangle,\varepsilon_4[1])}_{\varepsilon_5}\rangle} \text{(by (3.2))}.$$

Here

$$Q:=\underbrace{2:\mathbf{skip}}_{P_2} \parallel \underbrace{3:\mathbf{skip}}_{P_3}.$$

After this step, the environment stack array changes to ε_5 and S_1 becomes $\{S_{12}\}$. The current program becomes $1:\{S_{12}\} \parallel 2:\mathbf{skip} \parallel 3:\mathbf{skip}$.

Step 8 The assignment statement S_{12} of P_1 is to be executed and assignment statement rule (3.1) is applied. After the execution of the assignment statement, block rule (3.13) is applied.

$$\frac{\dfrac{[\![u]\!]_{\sigma_2}=5}{\pi_2 \mid \langle 1:v:=u \parallel Q\ ,\ \sigma_2,\ \varepsilon_5\rangle \longrightarrow \underbrace{push(\langle 1:v:=u,\ \sigma_2,\ \varepsilon_5\rangle,\pi_2)}_{\pi_3} \mid \langle 1:\mathbf{skip} \parallel Q,\ \underbrace{\sigma_2[5/v]}_{\sigma_3},\ \varepsilon_5\rangle} \text{(by (3.1))}}{\pi_2 \mid \langle 1:\{v:=u\} \parallel Q,\ \sigma_2,\ \varepsilon_5\rangle \longrightarrow \pi_3 \mid \langle 1:\{\mathbf{skip}\} \parallel Q,\sigma_3,\ \varepsilon_5\rangle} \text{(by (3.13))}.$$

Here

$$Q:=\underbrace{2:\mathbf{skip}}_{P_2} \parallel \underbrace{3:\mathbf{skip}}_{P_3}.$$

After this step, we get the final trace stack π_3 and the final state $\sigma_3=(o\mapsto 4,u\mapsto 5,v\mapsto 5)$. The current program becomes $1:\{\mathbf{skip}\} \parallel 2:\mathbf{skip} \parallel 3:\mathbf{skip}$.

Step 9 At last, block rule (3.11) is applied for the termination of the **if** statement S_1 of P_1.

$$\pi_3 \mid \langle 1:\{\mathbf{skip}\} \parallel 2:\mathbf{skip} \parallel 3:\mathbf{skip},\ \sigma_3,\ \varepsilon_5\rangle \longrightarrow \pi_3 \mid \langle 1:\mathbf{skip} \parallel 2:\mathbf{skip} \parallel 3:\mathbf{skip},\ \sigma_3,\ \underbrace{pop(\varepsilon_5[1])}_{\varepsilon_6}\rangle \text{ (by (3.11))}.$$

After this step, the environment stack array changes to ε_6 which is same as ε_0.