PAT: Language Syntax and Semantics

Jun Sun, Yang Liu and Jin Song Dong

School of Computing,
National University of Singapore
{sunj,liuyang,dongjs}@comp.nus.edu.sg

Abstract. Process Analysis Toolkit (PAT) is a toolset designed for specifying and verifying event-based compositional systems. In particular, it supports a variety of fairness assumptions. We explain in detail in this document the input language of PAT and its operational semantics.

1 Introduction

PAT is designed for systematic analysis of event-based compositional systems. The input language of PAT is mainly influenced by the classic Communicating Sequential Processes (CSP [Hoa85]). Nonetheless, we extend CSP with various useful language features to suit our goal. Examples include shared variables, asynchronous message passing channels and event annotations which capture a variety of fairness constraints. The language constructs may be categorized into the following groups.

- The first group is the core subset of CSP operators, including event-prefixing, internal/external choices, alphabetized lock-step synchronization, conditional branching, interrupt, recursion, etc.
- The second group includes those language constructs which can be regarded as 'syntactic sugar' (to CSP), including global shared variables, and asynchronous channels. It has long been known that CSP is capable of modeling shared variables or asynchronous channels as processes. However, the dedicated language constructs offer great usability and may make the verification more efficient.
- The third group is a set of event annotations for capturing event-based fairness constraints. It is known that process algebras like CSP or CCS specifies safety only. The event annotation offers a flexible way of modeling fairness using an event-based compositional language.
- The lastly group is the language for stating assertions, which later may be automatically verified using the built-in verifiers.

In the following, we will explain the language constructs one by one using illustrative examples. Each construct will be explained using a simple example, followed by informal explanation and formal operational semantics (for process constructs). Throughout the article, the dining philosophers example and the bridge crossing puzzle will be used as running examples.

Example 1 (Dining Philosophers).

[from Wikipedia] In 1971, Edsger Dijkstra set an examination question on a synchronization problem where five computers competed for access to five shared tape drive peripherals. Soon afterwards the problem was retold by Tony Hoare as the dining philosophers' problem.

Five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each philosopher, and as such, each philosopher has one fork to his or her left and one fork to his or her right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. The philosopher can only use the fork on his or her immediate left or right. It is further assumed that the philosophers are stubborn enough such that they only put down the forks after eating, and every philosopher picks up his left fork first and afterwards the right fork. There are a number of known problems with the dining philosophers. Firstly, the system may deadlock, i.e., all processes get stuck and cannot make a move. Secondly, the system may execute un-fairly. For instance, one philosopher may greedily grabs the forks and eats forever, leaving no chance at all for his neighbors. □

Example 2 (Bridge Crossing Puzzle). In the classic puzzle, there are four people arriving at an old bridge at night. All four people start out on the southern side of the bridge, namely the king, queen, a young lady and a knight. The goal is for everyone to arrive at the castle north of the bridge before the time runs out. The bridge can hold at most two people at a time and they must be carrying the torch when crossing the bridge. The king needs 5 minutes to cross, the queen 10 minutes, the lady 2 minutes and the night 1 minutes. The question is if the goal may be achieved given a specific time bound. □

2 Global Definitions

In PAT, a number of global definitions are used to assist elegant specification and verification, namely, global constants, variables, predicates (for assertions, refer to Section 5) and channels.

2.1 Global Constant

A global constant is defined using the following syntax,

$$\#define\ NoOfPhils\ 5;$$

$\#define$ is a key word used for multiple purposes. Here it defines a global constant named $NoOfPhils$, which has the value 5. The constant defines the size of the dining philosopher problem. The semi-colon marks the end of the 'sentence'. A global constant is cleared at compilation time (e.g., by replacing $NoOfPhils$ by 5 everywhere, refer to [SLD08]).

2.2 Global Variables

A global variable is defined using the following syntax,

$$knight = 0;$$

where $knight$ is the variable name. This variable tells whether the knight has reach the northern side of the bridge, i.e., 0 means no and 1 means yes. Semi-colon is used to mark the end of the 'sentence' as above. We remark that because only limited types are supported in PAT currently, no type information is required for a variable definition. An array may be defined as follows,

$$board = [3, 5, 6, 0, 2, 7, 8, 4, 1];$$

where $board$ is the array name and its initial value is specified as the sequence, e.g., $board[0] = 3$. The following defines an array of size 3.

$$leader[3];$$

All elements in the array are initialized to be 0.

2.3 Asynchronous Message Channels

A channel is defined using the following syntax,

$$channel\ c\ 5;$$

where $channel$ is a key word for defining channels only, c is the channel name and 5 is the buffer size of the channel. Notice that the buffer size must be a positive natural number. In SPIN, channels with buffer size 0 behaves as synchronization barriers. In PAT, synchronous communication is supported by CSP-style alphabetized parallel composition (refer to Section 3.14). A channel is implemented in PAT as a queue of the given size. In the following, we shall write $c.items$ to denote the items currently holding in channel c.

2.4 Aliasing for Conditions

A Boolean condition may be given a specific name (for referencing in the assertions, refer to Section 5) using the following syntax,

$$\#define\ goal\ (knight == 1 \&\& lady == 1 \&\& king == 1 \&\& queen == 1);$$

where $\#define$ is a key word (as used for defining global constants, refer to Section 2.1), and $goal$ is the name given for the Boolean expression

$$(knight == 1 \&\& lady == 1 \&\& king == 1 \&\& queen == 1)$$

which basically means that all four people have reached the northern side of the bridge.

3 Process Definitions

A process is defined as an equation in the following syntax

$$P(x_1, \cdots, x_n) = Exp;$$

where P is the process name, $x_1, \cdots, x_n$ is an optional list of process parameters and Exp is a process expression. In this section, the constructs for process expressions will be explained one-by-one. For each construct, a simple example is firstly presented to illustrate the syntax. Its formal semantics is then presented as a set of firing rules. A firing rule is of the following form,

$$\frac{enabling\ condition}{resultant\ transition}\ [\ rule\_name\ ]$$

The firing rules effectively define a labeled transition system semantics for a process expression.

Definition 1. *A Labeled Transition System (LTS) is a 4-tuple $(S, s_0, \rightarrow, \Sigma)$ where S is a set of states, $s_0 \in S$ is the initial state, $\rightarrow$: $S \times \Sigma \times S$ and Σ is the alphabet, i.e., a set of events.*

Given a process, its semantics is identified by an LTS $(S, s_0, \rightarrow, \Sigma$. S is a set of states of the form (p, v, c) where p is the current process expression, v is the current valuation of the global variables and c identifies the status of the channels. Notice that these three components uniquely identify the system state. s_0 is of the form (p_0, v_0, c_0), i.e., the initial process, the initial valuation and the state where all the channels are empty. $\rightarrow$ will be defined by the firing rules in a compositional manner. Σ (as we shall explain in detail, refer to Section 3.14) is the alphabet of the process, which may manually defined or it will be set to the default value.

3.1 Deadlock

The deadlock process is written as follows,

$Stop$

The process does nothing at all and thus there are no firing rules associated with $Stop$. The default alphabet of $Stop$ is $\varnothing$.

3.2 Termination

The process which terminates immediately[1] is written as follows,

$Skip$

Given V as the current valuation of the global variables and C as the status of the channels, the following firing rule is associated with the process $Skip$.

$$\frac{}{(Skip,\ V,\ C) \xrightarrow{\checkmark} (Stop,\ V,\ C)}\ [\ skip\ ]$$

✓ is a special event which denotes $termination$. The valuation of the variables and the status of the channels remain the same after the transition. Note that there were some known problem with ✓ with regards to parallel composition. Refer to our remedy presented in Section 3.14 and 3.13).

[1] PAT is only for un-timed systems at the moment and this 'immediately' shall not be interpreted as 'no time elaspes'.

3.3 Simple Event Prefixing

A simple event is a name for representing an observation. Given a process P, the following describes a process which performs e first and then behaves as specified by P,

$$e \rightarrow P$$

where e is an event.

$$\frac{}{(e \rightarrow P, V, C) \xrightarrow{e} (P, V, C)} [\ evtPref\ ]$$

The occurrence of a simple event does not modify the valuation of the global variables as well as the status of the channels. The following describes a simple vending machine which takes in a coin and dispatches a coffee every time.

$$VM() = insertcoin \rightarrow coffee \rightarrow VM();$$

where event $insertcoin$ models the event of inserting a coin into the machine and event $coffee$ models the event of getting coffee out of the machine. The following is one application of the above transition rule,

$$(insertcoin \rightarrow coffee \rightarrow VM(), V, C) \xrightarrow{insertcoin} (coffee \rightarrow VM(), V, C)$$

The same event may be shared by multiple processes running in parallel. The event then serves as a synchronization barrier (refer to Section 3.14). An event may be in a compound form, e.g., event $insert.50$ represents the event of inserting a coin of value 50 cents. Or an event may contain a local variable, as in the following example,

$$Phil(i) = get.i.(i+1)\%N \rightarrow Rest();$$

where i is a parameter of the process. Notice that the variables which constitute the event name will be replaced by its value whenever the process definition is instantiated (refer to [SLD08]).

Event e may be associated with a sequence of assignments, which update possibly multiple global variables at one time (refer to Section 3.4). Furthermore, events may be marked with special annotations to liveness/fairness (refer to Section 4).

Remarks Only process parameters may constitute the event name (not global variables). We make this assumption so that we may calculating the alphabet of a process meaningfully (refer to [SLD08]).

3.4 Assignments

The valuation of the global variables may be modified by assignments. A sequence of assignments may be attached to an event by one atomic operation, as in the following example,

$$go\_knight\_lady\{knight = 1;\ lady = 1;\ time = time + 2;\ \} \rightarrow North()$$

where go_knight_lady denotes the event of the knight and the lady crossing the bridge together (to the northern side, refer to Example 2). The variable $knight$ ($lady$) is 1 if and only if the knight (lady) has crossed the bridge. The variable $time$ is used to record the total time taken.

The assignments are evaluated in a sequence and therefore the order of the assignments does matter. The semantics of assignment is capture by the following rule,

$$\frac{}{(e\{assign\_exp\} \rightarrow P,\, V,\, C) \xrightarrow{e} (P,\, V',\, C)} [\ assign\ ]$$

where $assign\_exp$ is an assignment expression and V' is V with variable valuations updated accordingly to assignments[2].

Remarks In PAT, assignments are only allowed to update global variables but not local variables (e.g., parameters of the process definition). This is because local variables may constitute event names and therefore allowing dynamic modification of local variables may change the alphabet of a process dynamically, which is undesirable (refer to [SLD08]).

Remarks In general, a sequential program (which may involve while-loop, if-the-else, etc.) may be attached to an event. Because an event attached may update multiple global variables at once, synchronizing such events may result in race conditions. Therefore, events attached with assignments/programs are never synchronized.

3.5 Channel Input/Output

Channel input/output is written in a similar way to simple event prefixing.

$c!exp \rightarrow P$ – channel output
$c?x \rightarrow P$ – channel input

where c is a channel, exp is an expression which evaluates to a value and x is a (local) variable which takes the input value. The channel must be declared.

$$\frac{\neg\, full(c)}{(c!v \rightarrow P,\, V,\, C) \xrightarrow{c!v} (P,\, V,\, C \oplus (c.items \frown v))} [\ output\ ]$$

$$\frac{\neg\, empty(c),\, c.top = v,\, c' = c \frown v}{(c?x \rightarrow P,\, V,\, C) \xrightarrow{c?v} (P,\, V \oplus (x \mapsto v),\, C \setminus c \cup c')} [\ input\ ]$$

[2] We skip the semantics of the sequential composition of assignments.

3.6 External Choice

In PAT, we distinguish between $external$ choice and $internal$ choice. External choice is made by the environment, e.g., the observation of a visible event or the valuation of the global variables. Internal choice is used to model non-determinism explicitly (refer to Section 3.7). An external choice is written as follows,

$$Fork(x) = get.x.x \rightarrow put.x.x \rightarrow Fork(x) \ [] \\ get.(x-1)\%N.x \rightarrow put.(x-1)\%N.x \rightarrow Fork(x);$$

where event $get.x.y$ ($put.x.y$) is the event of x-philosopher picks up (put down) the y-fork. This process models the behavior of the fork in the dining philosopher example (refer to Example 1). The sub-process $get.x.x \rightarrow put.x.x \rightarrow Fork(x)$ says that the fork may be picked up by x-philosopher and later be put down by him/her. The sub-process $get.(x-1)\%N.x \rightarrow put.(x-1)\%N.x \rightarrow Fork(x)$ says that the fork may be picked up by the other philosopher and later be put down by him/her. The choice operator [] states that either philosopher may pick up the fork and once he/she does so, the other philosopher cannot do that any more. In other words, the choice is resolved by whichever (visible) event that occurs first.

The semantics of the external choice operators is captured by the following firing rules,

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \neq \tau}{(P \ [] \ Q, V, C) \xrightarrow{e} (P', V', C')} [\ ext_1\ ]$$

$$\frac{(P, V, C) \xrightarrow{\tau} (P', V', C')}{(P \ [] \ Q, V, C) \xrightarrow{e} (P' \ [] \ Q, V', C')} [\ ext_2\ ]$$

$$\frac{(Q, V, C) \xrightarrow{e} (P', V', C'), e \neq \tau}{(P \ [] \ Q, V, C) \xrightarrow{e} (Q', V', C')} [\ ext_3\ ]$$

$$\frac{(Q, V, C) \xrightarrow{\tau} (Q', V', C')}{(P \ [] \ Q, V, C) \xrightarrow{e} (P \ [] \ Q', V', C')} [\ ext_4\ ]$$

where τ denotes an internal (invisible) event. Rule ext_1 and ext_2 state that if a component may engage in a visible event e, then the choice is resolved if e is engaged. Rule ext_2 and ext_4 state that an invisible transition by either component does not resolve the choice. The following are the two possible transitions of $Fork(x)$,

$$Fork(x) \xrightarrow{get.x.x} put.x.x \rightarrow Fork(x)$$
$$Fork(x) \xrightarrow{get.(x-1)\%N.x} put.(x-1)\%N.x \rightarrow Fork(x)$$

3.7 Internal Choice

Internal choice introduces non-determinism explicitly. The following models a vending machine which produces one of two drinks non-deterministically,

$$nondetVM() = coin \rightarrow (tea \rightarrow nondetVM() <> coffee \rightarrow nondetVM());$$

The semantics of the internal choice is captured by the following firing rules.

$$\frac{}{(P <> Q, V, C) \xrightarrow{\tau} (P, V, C)} [\ int_1\ ]$$

$$\frac{}{(P <> Q, V, C) \xrightarrow{\tau} (Q, V, C)} [\ int_2\ ]$$

Both firing rules are un-conditional and therefore the system may take any transition.

Non-determinism is largely undesirable at design or implementation stage, whereas it is useful at modeling stage for hiding relevant information. For instance, it can be used to model the behaviors of a block-box procedure, where the exact details of the implementation is not available.

3.8 Conditional Choice

A choice may depend on a Boolean expression which in term depends on the valuations of the variables. In PAT, a number of different operators can be used to model conditional choices. Let P and Q be two process expressions.

$$coffee \rightarrow VM() << amount > 80 >> tea \rightarrow VM()$$

where $amount$ is the amount which has been inserted into the vending machine. Depending on the amount, one of the two drinks will be dispatched. In particular, if $amount$ is larger than 80, a coffee will be produced, otherwise a tea will be. The operator $<< b >>$ where b is a Boolean expression has the same semantics of if-then-else in programming languages like JAVA or C. The following capture its semantics,

$$\frac{V \vDash b, (P, V, C) \xrightarrow{e} (P', V', C')}{(P << b >> Q, V, C) \xrightarrow{e} (P', V', C')} [\ cond_1\ ]$$

$$\frac{V \nvDash b, (Q, V, C) \xrightarrow{e} (Q', V', C')}{(P << b >> Q, V, C) \xrightarrow{e} (Q', V', C')} [\ cond_2\ ]$$

PAT allows conditional choices to be written in the form of a guarded command, as in the following example.

$$[amount > 80]coffee \rightarrow VM()\ []$$
$$[amount <= 80 \&\& amount > 50]tea \rightarrow VM()$$

Intuitively, the above states that if more than 80 cents have been inserted into the vending machine, a coffee is dispatched; if the amount inserted is between 50 and 80, a tea is dispatched then. Process $[b]P$ is semantically equivalent to process $P << b >> Stop$.

$$\frac{V \vDash b, (P, V, C) \xrightarrow{e} (P', V', C')}{([b]P, V, C) \xrightarrow{e} (P', V', C')} [\ cond_3\ ]$$

The more general way of writing conditional choices is demonstrated in the following example,

$$\begin{aligned} Reading(i) = \mathit{if}\ & \\ & i == 0 : Controller() \\ & i == M : stopread \rightarrow Reading(i-1) \\ & i > 0 \&\& i < M : startread \rightarrow Reading(i+1)\ [] \\ & \qquad\qquad\qquad\quad stopread \rightarrow Reading(i-1) \\ & \mathit{endif}; \end{aligned}$$

Process $\mathit{if}\ (b_1 : P_1)\ (b_2 : P_2)\ \cdots\ (b_k : P_k)\ \mathit{endif};$ can be equivalently rewritten in the following form,

$$P_1 << b_1 >> P_2 << b_2 >> \cdots P_k << b_k >> Stop$$

That is, the conditions $b_1, \cdots, b_n$ are evaluated one by once in order until one process guarded by a condition which evaluated to true is found. Notice that internally, all other forms of conditional choices are translated to processes of the above form.

$$\begin{aligned} P << b >> Q &\equiv \mathit{if}\ (b : P)\ (\neg\, b : Q)\ \mathit{endif} \\ [b]P &\equiv \mathit{if}\ (b : P)\ \mathit{endif} \end{aligned}$$

3.9 Interrupt

Process $P\ |>\ Q$ behaves as specified by P until the first visible event of Q is engaged and then the control transfers to Q. The following is an example,

$$Routine()\ |>\ exception \rightarrow ExceptionHandling()$$

where $Routine$ is a process which performs normal daily task and $ExceptionHandling$ is a process which performs necessary actions for error handling. Whenever an exception occurs (modeled as event $exception$), process $ExceptionHandling$ takes the control over. The following rules capture its semantics,

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C')}{(P\ |>\ Q, V, C) \xrightarrow{e} (P'\ |>\ Q, V', C')} [\ interrupt_1\ ]$$

$$\frac{(Q, V, C) \xrightarrow{e} (P', V', C'), e \neq \tau}{(P\ |>\ Q, V, C) \xrightarrow{e} (Q', V', C')} [\ interrupt_2\ ]$$

$$\frac{(Q, V, C) \xrightarrow{\tau} (P', V', C')}{(P \mid> Q, V, C) \xrightarrow{\tau} (P \mid> Q', V', C')} [\ interrupt_3\ ]$$

An invisible transition of Q will not interrupt process P (rule $interrupt_3$). Notice that process $e \rightarrow Stop \mid> e \rightarrow Stop$ is trace-equivalent process $e \rightarrow Stop <> e \rightarrow Stop$.

3.10 Hiding

Process $P \setminus A$ where A is a set of events turns events in A to invisible ones. Hiding is applied when only certain events are interested. Hiding may be used to introduce non-determinism. In the following example, because event $get.i.i$ and $put.i.i$ are only relevant to process $Phil(i)$ and $Fork(i)$ in the dining philosopher problem, we may hide them from the rest of the system as in the following example.

$$\begin{array}{ll} Phil(i) & = get.(i+1)\%N.i \rightarrow get.i.i \rightarrow eat.i \rightarrow \\ & \quad put.(i+1)\%N.i \rightarrow put.i.i \rightarrow Phil(i); \\ Fork(i) & = get.i.i \rightarrow put.i.i \rightarrow Fork(i)\ [] \\ & \quad get.(i-1)\%N.i \rightarrow put.(i-1)\%N.i \rightarrow Fork(i); \\ PhilForkPair(i) & = (Phil(i) \parallel Fork(i)) \setminus \{get.i.i, put.i.i\} \end{array}$$

where $\parallel$ denotes alphabetized parallel composition (refer to Section 3.14). The following captures the semantics of the hiding operator.

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \in A}{(P \setminus A, V, C) \xrightarrow{\tau} (P' \setminus A, V', C')} [\ hiding_1\ ]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \notin A}{(P \setminus A, V, C) \xrightarrow{e} (P' \setminus A, V', C')} [\ hiding_2\ ]$$

A transition of P which is labeled with a visible event is turned into an τ-transition if the event is contained in A.

3.11 Selecting

The dual of hiding is *selecting*. Process P/A where A is a set of events turns events NOT in A to invisible ones. In contrast to hiding, A is the set of events which are of interest. This language feature is particularly useful when only a small number of events are of interest. Assume we are only interested in event $eat.i$ and do not care about the order in which the forks are picked up, we may hide other events from the rest of the system as in the following example.

$$\begin{array}{ll} PhilForkPair(i) & = (Phil(i) \parallel Fork(i)) \\ College() & = (PhilForkPair(0) \parallel PhilForkPair(1))/\{eat.0, eat.1\} \end{array}$$

where $\|$ denotes alphabetized parallel composition (refer to Section 3.14). The following captures the semantics of the hiding operator.

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \in A}{(P/A, V, C) \xrightarrow{e} (P'/A, V', C')} [\ selecting_1\ ]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \notin A}{(P/A, V, C) \xrightarrow{\tau} (P'/A, V', C')} [\ selecting_2\ ]$$

A transition of P which is labeled with a visible event is turned into an τ-transition if the event is not contained in A.

3.12 Sequential Composition

Sequential composition of two processes is written as $P;\ Q$, where $;$ serves as sequential composition operator. Intuitively, the semantics is that Q starts only when P terminates. The termination of P is signaled by the special event ✓ (refer to section 3.2). The following captures the semantics of sequential composition.

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \neq \checkmark}{(P;\ Q, V, C) \xrightarrow{e} (P';\ Q, V', C')} [\ seq_1\ ]$$

$$\frac{(P, V, C) \xrightarrow{\checkmark} (P', V', C')}{(P;\ Q, V, C) \xrightarrow{\tau} (Q, V', C')} [\ seq_2\ ]$$

Notice that whenever ✓ engages in P, Q takes over control and the system makes a τ-move. Though sequential composition seemed simple enough, it may not be so when mixed with parallel/interleaving composition (refer to Section 3.13 and 3.14).

3.13 Interleaving

Process $P \ ||| \ Q$ is the interleaving of the two processes P and Q. Thus, P and Q may behave independently of each other (with one exception, refer to the semantics). The following captures its semantics.

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \neq \checkmark}{(P \ ||| \ Q, V, C) \xrightarrow{e} (P' \ ||| \ Q, V', C')} [\ interleave_1\ ]$$

$$\frac{(Q, V, C) \xrightarrow{e} (Q', V', C'), e \neq \checkmark}{(P \ ||| \ Q, V, C) \xrightarrow{e} (P \ ||| \ Q', V', C')} [\ interleave_2\ ]$$

$$\frac{(P, V, C) \xrightarrow{\checkmark} (P', V, C), (Q, V, C) \xrightarrow{\checkmark} (Q', V, C)}{(P \;|||\; Q, V, C) \xrightarrow{\checkmark} (P' \;|||\; Q', V, C)} [\; interleave_3 \;]$$

The first two rules state that P or Q may behave on its own. The third rule states that P and Q must synchronize on termination. The intuition is that given two processes (which may be physically remote) running in parallel (with synchronization), the whole terminates if and only if both processes has terminated. For instance, given the process $(a \rightarrow \mathit{Skip} \;|||\; b \rightarrow \mathit{Skip});\; c \rightarrow \mathit{Stop}$, it is intuitive that c must happen only after both a and b have happened.

Remarks The indexed interleaving is written as

$$||| \; x : 0..N \bullet P(x)$$

where N is a natural number. Given N is of value 2, the above process is compiled to $P(0) \;|||\; P(1) \;|||\; P(2)$.

3.14 Parallel Composition

Parallel composition of two processes with synchronization is written as

$$P \parallel Q$$

where $\parallel$ is the parallel composition operator. In parallel composition, abstract events shared by both P and Q must be synchronized.

Before we explain the semantic rules, we must first define the alphabet of a process. The alphabet of a process is the set of events that the process takes part in. For instance, given the process defined as follows,

$$VM() = \mathit{insertcoin} \rightarrow \mathit{coffee} \rightarrow VM();$$

The alphabet of $VM()$ is exactly the set of events which constitute the process expression, i.e., $\{\mathit{insertcoin}, \mathit{coffee}\}$. However, calculating the alphabet of a process is not always trivial. It may be complicated by two things. One is process referencing. The other is process parameters. In the above example, the process reference $VM()$ happens to be the same as the process whose alphabet is being calculated. Thus, it is not necessarily to unfold $VM()$ again. Should a different process is referenced, we must unfold that process and get its alphabet. For instance, assume the $VM()$ is now defined as follows,

$$\begin{aligned} VM() &= \mathit{insertcoin} \rightarrow \mathit{Inserted}(); \\ \mathit{Inserted}() &= \mathit{coffee} \rightarrow VM(); \end{aligned}$$

To calculate the alphabet of $VM()$, we must unfold process $\mathit{Inserted}()$ and combine alphabets of $\mathit{Inserted}()$ with $\mathit{insertcoin}$. Notice that a simple procedure must be used to prevent unfolding the same process again. However, even with such a procedure, it may

still be infeasible to calculate mechanically the alphabet of a process. The complicated is due to process parameters. For instance, given the following process,

$$P(i) = a.i \rightarrow P(i+1);$$

Naturally, the unfolding process is non-terminating. In general, there is no way to solve this problem. Therefore, PAT offers two compromising ways to get the alphabets. One is to use a reasonably simple procedure to calculate a default alphabet of a process. When the default alphabet is not as expected, an advanced user is allowed to define the alphabet of a process manually. We detail the first in the following.

First of all, alphabet of processes are calculated only when it is necessarily. That means, only when a parallel composition is evaluated. This saves a lot of computational overhead. Processes in a large number of models only communicate through shared variables. If no parallel composition is present, there is no need to evaluate alphabets. We remark that when there is no shared abstract events, process $P \ ||| \ Q$ and $P \parallel Q$ are exactly the same. Therefore, we recommend $|||$ when appropriate. When a parallel composition is evaluated for the first time, the default alphabet of each sub-process is calculated (if not manually defined). **The default alphabet of a process is the set of events which constitute its process expression. If process references are present, each process reference is unfolded exactly once**. By far, this default alphabet has been intuitive.

The following is the firing rules associated with parallel composition. Let Σ_P be the alphabet of process P.

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), e \notin \Sigma_Q}{(P \parallel Q, V, C) \xrightarrow{e} (P' \parallel Q, V', C')} [\ parallel_1\ ]$$

$$\frac{(Q, V, C) \xrightarrow{e} (Q', V', C'), e \notin \Sigma_P}{(P \parallel Q, V, C) \xrightarrow{e} (P \parallel Q', V', C')} [\ parallel_2\ ]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V, C), (Q, V, C) \xrightarrow{e} (Q', V, C)}{(P \ ||| \ Q, V, C) \xrightarrow{e} (P' \parallel Q', V, C)} [\ parallel_3\ ]$$

The alphabet of a process always implicitly includes the special event ✓. Therefore, by the last rule above, ✓ as other shared events must be synchronized. Notice that the last rule assumes that the synchronization does not change the global variables or channels, which is guaranteed in PAT.

3.15 Process Referencing

We have seen a lot of process referencing by far. Process referencing is used to invoke an already-defined process, with customized parameters. It is also used to realize recursive processes or mutual recursive processes.

Annotation	Name	Semantics
$wf(e)$	weak fair event	$\Diamond\Box e$ is enabled $\Rightarrow \Box\Diamond e$ is engaged
$sf(e)$	strong fair event	$\Box\Diamond e$ is enabled $\Rightarrow \Box\Diamond e$ is engaged
$wl(e)$	weak live event	$\Diamond\Box e$ is ready $\Rightarrow \Box\Diamond e$ is engaged
$sl(e)$	strong live event	$\Box\Diamond e$ is ready $\Rightarrow \Box\Diamond e$ is engaged

Table 1. Event-based Fairness Annotations

4 Event-based Fairness

PAT support the notion of event-based fairness, i.e., fairness constraints which are associated with individual events. Given any event e, four different annotations can be used to associate different fairness assumptions with e. The annotations are summarized in Table 1. In the following, we discuss them one by one.

4.1 Weak Fair Events

A *weak* fair event is written as

$$wf(e)$$

Event $wf(e)$ plays the same role as e except that it carries a weak fairness constraint. That is, if a weak fair event is always enabled, it must be eventually engaged. In other words, the system must move beyond a state where there is a weak fair event enabled. Weak fair events allow us to express weak fairness constraints naturally. The following demonstrates how we may achieve process level weak fairness (as the option offered in SPIN) using weak fair events.

$$\begin{aligned} \mathit{fPhil}(i) &= wf(get.i.(i+1)\%N) \rightarrow wf(get.i.i) \rightarrow wf(eat.i) \\ &\quad \rightarrow wf(put.i.(i+1)\%N) \rightarrow wf(put.i.i) \rightarrow \mathit{fPhil}(i) \\ \mathit{fFork}(i) &= wf(get.i.i) \rightarrow wf(put.i.i) \rightarrow \mathit{fFork}(i) \,[]\, \\ &\quad wf(get.(i-1)\%N.i) \rightarrow wf(put.(i-1)\%N.i) \rightarrow \mathit{fFork}(i) \\ \mathit{fCollege}() &= \Big\| \, x : \{0..N-1\} @ (\mathit{fPhil}(x) \parallel \mathit{fFork}(x)) \end{aligned}$$

The idea is to annotate all events in a process weak fair so that an enabled event of the process is not ignored forever.

4.2 Strong Fair Events

It can be shown that both weak and strong fairness are expressible using weak fair events (as strong fairness can be transformed to weak fairness by paying the price of one variable). However, strong fairness constraints may require more than what fair events can offer in a natural way. Therefore, we introduce the notion of *strong* fair events to capture strong fairness elegantly. A *strong* fair event, written as $sf(e)$, must be engaged if it is repeatedly enabled.

The following specifies the Peterson's algorithm for mutual exclusion. Without fairness assumptions, the algorithm allows unbounded overtaking, i.e., a process which intends to enter the critical section may be overtaken by other processes infinitely.

$$P(i,j) = (sf(update.i.j)\{pos[i] := j;\ step[j] := i;\ \} \rightarrow Rest()$$

where $pos, step$ are two lists of integers (with initial value 0) of size $N-1$ and N respectively. In order to guarantee a system is completely strongly fair, communicating events or events guarded by conditions must be annotated with strong fairness, whereas weak fairness is sufficient for local actions which are not guarded.

4.3 Live Events

The following two event annotations are connected to CSP's alphabetized parallel synchronization. Skip this section if necessary.

In practice, even stronger fairness may be necessary. One example of a fairness constraint which is very strong is the notion of accepting states in Büchi automata, i.e., the system must keep moving until entering at least one accepting state (and do that infinitely often). Other examples of stronger fairness include the compassion conditions [KPRS06]. In order to capture these fairness constraints, we introduce two additional fairness annotations, which have the capability of driving the system to reach certain point. The additional annotations relies on the concept of "readiness" so that system behaviors may be restricted by fairness assumptions which are associated with events that are not even enabled.

Definition 2 (Readiness). *Let P be a process. Let V be a valuation of the variables.*

$$\begin{array}{lll}
ready(Stop, V) & = ready(Skip, V) = \varnothing & \\
ready(e \rightarrow P, V) & = \{e\} & \\
ready(Skip;\ Q, V) & = ready(Q, V) & \\
ready(P;\ Q, V) & = ready(P, V) & - \textit{if } P \neq Skip. \\
ready(P [] Q, V) & = ready(P, V) \cup ready(Q, V) & \\
ready(P <> Q, V) & = ready(P, V) \cup ready(Q, V) & \\
ready(P \mid> Q, V) & = ready(P, V) \cup ready(Q, V) & \\
ready(P << b >> Q, V) & = ready(P, V) & - \textit{if } V \vDash b. \\
ready(P << b >> Q, V) & = ready(Q, V) & - \textit{if } V \vDash \neg\, b. \\
ready(P \parallel Q, V) & = ready(P, V) \cup ready(Q, V) &
\end{array}$$

Event e is *ready* given process P and valuation V if and only if $e \in ready(P, V)$. Note that *enabledness* and *readiness* are similarly defined for all process expressions except parallel composition. The difference is captured by the last line of the above definition. Given process $P \parallel Q$, a shared event is enabled if and only if it is enabled in both P and Q, whereas it is ready if it is ready in either P or Q. Intuitively, an event is *ready* if and only if one thread of control is ready to engage in it. An *enabled* event must be *ready*. A *weak* live event, written as $wl(e)$, must be engaged if it is always ready (not necessarily enabled). Similarly, a *strong* live event, written as $sl(e)$, must be engaged if it is repeated ready. Because whether an event is ready or not depends on only one

process (in a parallel composition), live events may be used to design a controller which drives the execution of a given system.

Let $LiftSystem()$ be the modeling of a multi-lift system, which contains two events $turn\_on\_light$ and $turn\_off\_light$. In order to model that the light is always eventually turned off, the $LiftSystem()$ may be replaced by $LightSystem() \parallel LightCon()$ where $LightCon() = turn\_on\_light \rightarrow wl(turn\_off\_light) \rightarrow LightCon()$. Because both events must be synchronized, whenever event $turn\_on\_light$ is engaged, event $turn\_off\_light$ becomes ready. In this case, it remains ready until it is engaged. Thus, by definition, the light must eventually be turned off.

With live events, the dining philosophers may be modified as follows,

$$
\begin{array}{ll}
lPhil(i) & = wl(get.i.(i+1)\%N) \rightarrow get.i.i \rightarrow eat.i \\
 & \quad \rightarrow put.i.(i+1)\%N \rightarrow put.i.i \rightarrow lPhil(i) \\
lFork(i) & = get.i.i \rightarrow wl(put.i.i) \rightarrow lFork(i)[] \\
 & \quad get.(i-1)\%N.i \rightarrow wl(put.(i-1)\%N.i) \rightarrow lFork(i) \\
lCollege(N) & = \Big\| \; x : \{0..N-1\}@(lPhil(x) \parallel lFork(x))
\end{array}
$$

Model checking $\Box\Diamond eat.0$ against $lCollege(5)$ returns true. Initially, $wl(get.i.(i+1)\%N)$ is ready and therefore by definition, it must be engaged (since it is not possible to make it not ready). Once $get.i.(i+1)\%N$ is engaged, $wl(put.(i-1)\%N.i)$ becomes ready and thus the system is forced to execute until it is engaged. For the same reason, $wl(put.i.i)$ must be engaged afterwards. Once $put.i.i$ is engaged, $wl(get.i.(i+1)\%N)$ becomes ready again. Therefore, the system is forced to execute infinitely and fairly. The traces which lead to the deadlock state is not returned as a counterexample. This is because event $wl(put.(i-1)\%N.i)$ is ready in the deadlock state. Hence the trace is considered invalid because it does not satisfied the fairness assumption, i.e., the event $wl(put.(i-1)\%N.i)$ is always ready but never engaged.

The fairness annotations restrict the possible behaviors of the system. It disallows unfair (or unrealistic) traces and results in a smaller set of traces. Note that fairness constraints cannot be captured using firing rules. Therefore, a two-levels semantics is used to prune un-fair traces from infinite traces. The semantics is embedded in the verification algorithms implemented in PAT (refer to [SLD08]).

5 Assertions

PAT allows user to state a number of different assertions. We details them one-by-one in the following.

5.1 Deadlock-freeness

To assert that that a process is deadlock-free, we write the following assertion,

$$\#assert\ College()\ deadlockfree;$$

A counter example to the assertion is a finite sequence of events which lead to a deadlock state.

5.2 Reachability Analysis

Reachability analysis is used to verify whether a particular condition could be true or not. It is often useful for solving puzzles. For instance, given process $BridgeCrossing()$ which models the bridge crossing puzzle, the following asks for a solution.

$$\#assert\ BridgeCrossing()\ reachable\ goal;$$

where $goal$ is a aliasing for the goal configuration of the game, i.e., all crossed the bridge (refer to Section 2.4). Intuitively, the assertion states that a state satisfying the $goal$ condition is reachable in process $BridgeCrossing()$.

5.3 Linear Temporal Logic

PAT allows the full set of LTL. An LTL assertion is of the following form.

$$\#assert\ P()\ |=\ LTLFormula$$

where $P()$ is a process and $LTLFormula$ is an LTL formula.

To verify that something always holds, e.g., the balance of a saving account is always positive, the following LTL formula is used, assuming that $balance$ is the variable recording the balance of the account.

$$\#define\ inv\ balance\ >=\ 0;$$
$$\#assert\ Account()\ |=\ []inv;$$

where inv is defined as an aliasing for the predicate $balance\ >=\ 0$ (refer to Section 2.4). The modal operator [] reads as 'always'. There are a number of other modal operators. The following shows a list using illustrative examples. Assume F, H are LTL formulae,

$[]F$	– always F is true.
$<>\ F$	– eventually F is true.
$F\ \&\&\ H$	– F and H are true.
$F \rightarrow H$	– F implies H.
$X\ F$	– nextly F is true.

Notice that it is possible to compose different modal operators, for instance $[]\ <>$ is often used to 'always eventually' something happens, or $<>\ []$ to 'eventually always' something happens, etc.

5.4 Refinement and Equivalence

The following assertions are used to check whether one process refines/equals the other. There are three different kinds of refinement (equivalence) relationship between two processes, namely trace refinement (equivalence), stable failures refinement (equivalence) and failures/divergence refinement (equivalence). Refer [Hoa85,For03] for detailed introduction on these notions. Intuitively, in contrast to LTL formulae which talk

about properties of each execution of a given process, refinement (equivalence) checking looks at the behaviors of a process in whole and compare them with the behaviors of another process.

$$\#assert\ Implementation()\ [T=\ Specification();$$
$$\#assert\ Specification()\ [T=\ Implementation();$$
$$\#assert\ Implementation()\ [F=\ Specification();$$
$$\#assert\ Specification()\ [F=\ Implementation();$$
$$\#assert\ Implementation()\ [FD=\ Specification();$$
$$\#assert\ Specification()\ [FD=\ Implementation();$$

References

[For03] Formal System Europe. Failure Divergence Refinement. http://www.fsel.com/, 2003.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[KPRS06] Y. Kesten, A. Pnueli, L. Raviv, and E. Shahar. Model Checking with Strong Fairness. *Formal Methods and System Design*, 28(1):57–84, 2006.

[SLD08] Jun Sun, Yang Liu, and Jin Song Dong. PAT Reference Manual: Verification Algorithms. Technical report, 2008.