Verifying Mutual Exclusion and Liveness Properties with Split Preconditions

Awadhesh Kumar Singh[1] and Anup Kumar Bandyopadhyay[2]

[1]*Department of Computer Engineering, National Institute of Technology, Kurukshetra 136119, India*

[2]*Department of Electronics and Telecommunication Engineering, Jadavpur University, Kolkata 700032, India*

E-mail: aksinreck@rediffmail.com; anupbandyopadhyay@hotmail.com

Received September 5, 2003; revised May 30, 2004.

Abstract This work is focused on presenting a split precondition approach for the modeling and proving the correctness of distributed algorithms. Formal specification and precise analysis of Peterson's distributed mutual exclusion algorithm for two process has been considered. The proof of properties like, mutual exclusion, liveness, and lockout-freedom have also been presented.

Keywords distributed algorithms, state transition rule, mutual exclusion, weakest self-precondition, weakest co-operation, correctness

1 Introduction

Distributed algorithms run on machines having a collection of interconnected processors. These processors communicate in various unpredictable ways. This makes the environment more complicated. Therefore, the issue of prime concern is the correct execution of the distributed algorithms, irrespective of the order of interleaving of concurrent processes. This problem can be taken care of by proper modeling and thorough analysis of properties of distributed algorithms. This analysis can be carried out statically or dynamically. The dynamic analysis relies on the limited number of test runs to make observations about the behavior of the distributed algorithm. A couple of tools[1−4] are available to perform this task. However, since the dynamic techniques do not take into consideration all possible executions, therefore, they are not sound[5]. Thus, we resort to static analysis, though it involves considerable amount of human effort. Although a large number of techniques[6−19] are available in the literature, the static analysis is difficult to carry out even for a simple distributed algorithm. If we consider a complex algorithm, the complicacy of this task is enhanced further, in absence of some formal and precise method for reasoning[20]. Therefore, the objective of this work is to present a new approach, hereafter we say split precondition approach, which is simple to understand and easy to apply, for assertional reasoning of distributed algorithms. In order to assert this, the proof of all properties of Peterson's algorithm[21] and rigorous formalization of the proof has been presented.

Many previous researchers exhaustively analyzed the properties of Peterson's algorithm. An early effort, to verify it, was made by Alpern and Schneider[22] without temporal logic and based on Buchi automata[23]. This approach, despite not being limited to finite state spaces, requires creativity in devising the proof instruments like, an invariant to handle safety aspects, a variant function to handle liveness aspects, and a candidate function. Moreover, it is not always guaranteed to get the correct answer[22]. Later, Shankar, in his tutorial[24], tried to prove Peterson's algorithm with assertional reasoning. Manna and Pnueli[25] did it in the frame of linear time temporal logic and Pau *et al.*[20] applied TLA (temporal logic of actions) for the same purpose. Although the details of their proofs are different, all approaches are similar. They carried out proofs by formulating assertions, i.e., by stating properties about all executions of the system. Each assertion in the sequence is proved by the application of some proof rule. The major problem of such an approach is that it is considered "difficult" and a common complaint about such proofs is that they are too tedious[26] and the insight gained by explicitly considering sequences of events is often lost or buried among the details of the proof[27]. The recent approach, by Win and Ernst[5], uses dynamic and static analysis as complement to each other. They have made few case studies including verification of Peterson's algorithm. Their approach relies on some verification tools like I/O automaton model[28,29], IOA

*Regular Paper

language[30], Larch Prover[31], and the Daikon dynamic invariant detector[32]. Whether the use of these tools can be generalized more broadly, is not yet established.

Our split precondition logic uses First Order Predicate Logic (FOPL), which is very easy to apply. In this approach, we write the formulas and prove them logically. Moreover, this approach is better than Chandi-Sanders' approach[33] as in Chandi-Sanders' approach co-operation requirement is to be catered separately, whereas in our approach, it is inbuilt. The idea, of splitting one precondition into two entities, is borrowed from [34] where authors used it to detect deadlock condition in a distributed system. We propose to extend the proof technology into the realm of modeling and reasoning about the correctness of distributed algorithms. It has been illustrated by considering well-known Peterson's algorithm[21].

Use of split precondition approach contributes two essential ingredients to the analysis of distributed algorithms. Firstly it provides an elegant and rigorous notation for describing distributed algorithms. Secondly, the split precondition approach provides a framework for the systematic analysis of the correctness of distributed algorithms.

This article is not simply about representing a known algorithm; it also aims to demonstrate that the split precondition logic is suitable for both modeling and reasoning about a distributed algorithm. Our hope is that the approach used here could be applied fruitfully to other distributed algorithms.

2 Description of the Split Precondition Logic

Like every formal language, split precondition logic also has a well-defined syntax and semantics. The description of the mathematical model is being given in the following paragraphs.

A set of states $P.X$ and a set of state transition rules $P.R$ define a process P. On the similar lines a set of processes $S.P$ interacting through message transactions define a system S. The expression $in(P.x)$ represents that a process P is in state $P.x$. The initial state of process, which is predefined, is denoted by the expression $initial(P.x_0)$. The collection of states of all the processes belonging to set $S.P$ is termed as state SX of the system S.

A state transition rule represents the movement of process from one state to another. In order to fire any transition rule $P.r$ and eventual establishment of postcondition Q, there exists a corresponding weakest precondition $wp(P.r, Q)$. We postulate the value of wp for a specific post condition Q and for a specific transition $P.r$. Though the name, weakest precondition, is similar, but our wp should not be confused with the wp used by Dijkstra[35]. He used it to represent a predicate transformer, i.e., a function that maps any predicate (that expresses a required post condition) to another predicate (that expresses the precondition for the statement to terminate in a state that satisfies the post condition). Hence, in split precondition logic, unlike Dijkstra logic, weakest preconditions are not computed from the program text, but rather they provide a new model of the program, which reflects the original semantics of the program. If the system state satisfies $wp(P.r, Q)$ then execution of $P.r$ will eventually establish the post condition Q. However, this is not guaranteed unless $wp(P.r, Q)$ is true before the execution of $P.r$. The "precondition" is "weakest" in this sense only. In our logic, weakest precondition has been split into two entities:

(i) $wsp(P.r, Q)$, termed as the weakest self precondition and is related to the process P itself;

(ii) $wcr(P.r, Q)$, termed as the weakest co-operation requirements and includes the co-operation requirements from other processes.

Thus, the total weakest precondition will be given by

$$wp(P.r, Q) = wsp(P.r, Q) \wedge wcr(P.r, Q).$$

This justifies the appropriateness of the name split precondition logic. Since, the co-operation requirements have already been included in the wp in our approach, separate proof of co-operation, as required in [33], is not necessary here. Any transition rule $P.r$ is described jointly by the weakest precondition $wp(P.r, Q)$ and the post condition Q. This scheme is illustrated in the following example.

Example. A system S is described by the following informal specification:

1) there are two processes P_1 and P_2 in S;

2) a message transfer occurs from P_1 to P_2 when:

(a) an input command with P_1 as source is executed in P_2;

(b) an output command with P_2 as destination is executed in P_1;

3) a process must wait until the other process is ready for message transaction, in other words the input and output command must be synchronized.

2.1 Formal System Specification

2.1.1 *States of Process P_1*

State	Semantics
1. $P_1.beo$	state of P_1 before execution of the output command;
2. $P_1.rts$	state of P_1 when it is ready to send a message. This state occurs just after the execution of the output command;
3. $P_1.sent$	state of P_1 describing the fact that the message transaction is over.

2.1.2 *States of Process P_2*

State	Semantics
1. $P_2.bei$	state of P_2 before execution of the input command;
2. $P_2.rtr$	state of P_2 when it is ready to receive a message. This state occurs just after the execution of the input command;
3. $P_2.rec$	state of P_2 describing the fact that the message transaction is over.

With reference to the above states we can describe the processes as follows.

Process P_1; identified by P_1;
States: $P_1.beo$, $P_1.rts$, $P_1.sent$;

2.1.3 *Transition Rules*

$P_1.r_1$ ::
$wsp(P_1.r_1,\ in(P_1.rts)) = in(P_1.beo)$
$wcr(P_1.r_1, in(P_1.rts)) = true$
end of $P_1.r_1$;
$P_1.r_2$ ::
def $b = in(P_1.sent) \wedge in(P_2.rec)$
$wsp(P_1.r_2, b) = in(P_1.rts)$
$wcr(P_1.r_2, b) = in(P_2.rtr)$
end of $P_1.r_2$;
end of transition rules;
Initial state: $in(P_1.beo)$
end of process P_1.

Process P_2; identified by P_2;
States: $P_2.bei$, $P_2.rtr$, $P_2.rec$;

2.1.4 *Transition Rules*

$P_2.r_1$ ::
$wsp(P_2.r_1, in(P_2.rtr)) = in(P_2.bei)$
$wcr(P_2.r_1, in(P_2.rtr)) = true$
end of $P_2.r_1$;
$P_2.r_2$ ::
def $b = in(P_1.sent) \wedge in(P_2.rec)$
$wsp(P_2.r_2, b) = in(P_2.rtr)$
$wcr(P_2.r_2, b) = in(P_1.rts)$
end of $P_2.r_2$;
end of transition rules.
Initial state: $in(P_2.bei)$;
end of process P_2.

The transition rule $P_1.r_2$ causes state transitions for both the processes P_1 and P_2. Hence, a corresponding transition rule, viz., $P_2.r_2$ is also defined in the process P_2. Though message transaction can occur between two processes only, we may have to include assertions about the states of more than one processes in $wcr(P.r, Q)$. This is necessary when P is allowed to accept a message from P_i only when P_j has not invoked a send command. Implementation of such a system is possible by using a control structure like pri-alt in Occam.

2.2 Non-Deterministic State Transition Rule

A non-deterministic state transition rule $P.r$ may include a number of different sub-rules each of which requires a definite precondition to be satisfied for its execution. These preconditions will be called guards. Execution of a sub-rule will change the state of P as well as the state of one of the co-operating processes whose active co-operation is necessary for this execution. State transition in the co-operating process will be achieved by simultaneous execution of a state transition rule. If the preconditions for more than one sub-rule are satisfied then one of them is selected for execution. Selection procedure is non-deterministic and therefore, it is necessary to pass this information to the relevant co-operating process to produce the required state transition. The weakest precondition for a non-deterministic transition rule $P.r$ is obtained as follows.

Let there be n number of sub-rules denoted by $P.r^i$: $i = 1, \ldots, n$. On the top of these sub-rules we assume a selector procedure, denoted by *select*, which makes the required non-deterministic selection. The post condition space for this procedure should therefore include a number of Boolean variables denoted by s_i: $i = 1, \ldots, m$. At each invocation the selector makes one such variable true. If a sub-rule $P.r^i$ has a post condition Q_i then

$$s_i \Rightarrow wp(P.r^i, Q_i).$$

Let B_i denote the required guard for $P.r^i$, then the truth of this condition should ensure the selectability of s_i, i.e.,

$$B_i \Rightarrow wr(select, s_i),$$

where $wr(select, s_i)$ is the weakest requirement that the procedure *select* may produce s_i. Using equations for s_i and B_i the rule $P.r$ is described as follows:

```
$P.r$ ::
$Q \equiv \exists i\colon\ Q_i$;
$wp(P.r, Q) = (\exists k\colon\ B_k) \wedge$
        $(\forall i \cdot B_i \Rightarrow wr(select, s_i)) \wedge$
        $(\forall j \cdot s_j \Rightarrow wp(P.r^j, Q_j))$;
end of $P.r$;
```

2.3 Property of a System

The operational model of a system can be described by state transition rules. These rules can be described completely by their weakest precondition, post condition pairs. However, only operational specification may not be sufficient to describe the system requirements. In order to specify a system completely, along with the state transition rules the system properties must also be explicitly described. Best way to do this is to define a system invariant, which must remain true before and after the execution of each state transition rule. That is, there must exist a condition Q such that

$$\forall i \cdot \forall m\{wp(P_i.r_m, Q_i, m) \Rightarrow Q\} \wedge (Q_i, m \Rightarrow Q).$$

Similarly for a guarded command we have

$$\forall i \cdot (B_i \Rightarrow Q) \wedge (B_i \Rightarrow wp(P.r^i, Q_i)) \wedge (Q_i \Rightarrow Q).$$

3 Principle of Peterson's Two-Process Mutual Exclusion Algorithm

Peterson[21] gave a very simple solution to the mutual exclusion problem involving two processes, 1 and 2. It resolves simultaneity conflicts for a shared resource like, printer, data structure, etc., and if a process wishes to access the shared resource then it will succeed eventually.

The solution uses three shared variables *flag*$_1$, *flag*$_2$, and *turn*, readable by both processes. The variable *flag*$_1$ and *flag*$_2$ are writable only by Process 1 and Process 2 respectively whereas the variable *turn* is writable by both processes. The variables *flag*$_1$ and *flag*$_2$ are binary. Their value "1" means corresponding process is interested to enter its critical section and the process shows disinterest by setting the value "0". In order to make a request to enter its critical section the process sets its flag to "1" and maintains this value until the process comes out of its critical section. The variable *turn* assumes values from $\{1, 2\}$ and is used to resolve contention. A process can access the critical section only when the other process either does not wish to or makes request later.

4 Algorithm

```
cobegin
    repeat
      $flag_1$:=1
      $turn$:=1
      waitfor $flag_2 = 0$ or $turn = 2$
      (*critical section*)
      $flag_1$:=0
      (*non critical section*)
    forever
||
    repeat
      $flag_2$:=1
      $turn$:=2
      waitfor $flag_1 = 0$ or $turn = 1$
      (*critical section*)
      $flag_2$:=0
      (*non critical section*)
    forever
cobegin
```

In the algorithm "non critical section" refers to the execution of some command when the process has released the shared resource and has not requested for it again.

5 Formalization of the Algorithm

Now, we formalize the above-described distributed algorithm using split precondition technique. Like in other modeling techniques, we also make certain assumptions, which provide framework for analysis of the algorithm. Each process executes at non-zero speed but we make no assumption on the relative speed of processes. Several CPUs may be present but memory hardware prevents simultaneous access to the same memory location. We also make no assumption about the order of interleaved execution. Our technique also views the execution of the distributed algorithm in terms of the atomic events. These events are communication with the other process, that is, access to shared variable. Due to our assumption regarding atomicity, we can formalize the distributed algorithm as a state transition system with two processes as P_1 and P_2.

5.1 States of Process P_1

	State	Semantics
1.	$(P_1.ncs)$	P_1 is in non critical section
2.	$(P_1.flag_1 = 1)$	P_1 has set local variable *flag*$_1$
3.	$(P_1.turn = 1)$	P_1 has set shared variable *turn* favorably
4.	$(P_1.wait\_for)$	P_1 is waiting for favorable condition to enter critical section
5.	$(P_1.enter\_cs)$	P_1 has entered critical section
6.	$(P_1.flag_1 = 0)$	P_1 has reset local variable *flag*$_1$

5.2 States of Process P_2

	State	Semantics
1.	$(P_2.ncs)$	P_2 is in non critical section
2.	$(P_2.flag_2 = 1)$	P_2 has set local variable *flag*$_2$
3.	$(P_2.turn = 2)$	P_2 has set shared variable *turn* favorably
4.	$(P_2.wait\_for)$	P_2 is waiting for favorable condition to enter critical section
5.	$(P_2.enter\_cs)$	P_2 has entered critical section
6.	$(P_2.flag_2 = 0)$	P_2 has reset local variable *flag*$_2$

5.3 Transition Rules

Process P_1**;** identified by P_1
Transition rules for process P_1;

$P_1.r_1$ :: % Set flag %
$wsp(P_1.r_1,\ in(P_1.flag_1 = 1)) = in(P_1.ncs)$
$wcr(P_1.r_1, in(P_1.flag_1 = 1)) = true$
end of $P_1.r_1$;

$P_1.r_2$ :: % Set turn %
$wsp(P_1.r_2, in(P_1.trun = 1)) = in(P_1.flag_1 = 1)$
$wcr(P_1.r_2, in(P_1.turn = 1)) = true$
end of $P_1.r_2$;

$P_1.r_3$ :: % Wait %
$wsp(P_1.r_3, in(P_1.wait\_for)) = in(P_1.turn_1 = 1)$
$wcr(P_1.r_3, in(P_1.wait\_for)) = in(P_2.flag_2 = 1)$
end of $P_1.r_3$;

$P_1.r_4$ :: % Execute critical section %
$wsp(P_1.r_4, in(P_1.enter\_cs)) = in(P_1.wait\_for)$
$wcr(P_1.r_4, in(P_1.enter\_cs)) = in(P_2.flag_2 = 0) \vee in(P_2.turn = 2)$
end of $P_1.r_4$;

$P_1.r_5$ :: % Release critical section %
$wsp(P_1.r_5, in(P_1.flag_1 = 0)) = in(P_1.enter\_cs)$
$wcr(P_1.r_5, in(P_1.flag_1 = 0)) = true$
end of $P_1.r_5$;

$P_1.r_6$ :: % Execute non critical section %
$wsp(P_1.r_6, in(P_1.ncs)) = in(P_1.flag_1 = 0)$
$wcr(P_1.r_6, in(P_1.ncs)) = true$
end of $P_1.r_6$;

Process P_2**;** identified by P_2
Transition rules for process P_2;

$P_2.r_1$ :: % Set flag %
$wsp(P_2.r_1, in(P_2.flag_2 = 1)) = in(P_2.ncs)$
$wcr(P_2.r_1, in(P_2.flag_2 = 1)) = true$
end of $P_2.r_1$;

$P_2.r_2$ :: % Set turn %
$wsp(P_2.r_2, in(P_2.trun = 2)) = in(P_2.flag_2 = 1)$
$wcr(P_2.r_2, in(P_2.turn = 2)) = true$
end of $P_2.r_2$;

$P_2.r_3$ :: % Wait %
$wsp(P_2.r_3, in(P_2.wait\_for)) = in(P_2.turn = 2)$
$wcr(P_2.r_3, in(P_2.wait\_for)) = in(P_1.flag_1 = 1)$
end of $P_2.r_3$;

$P_2.r_4$ :: % Execute critical section %
$wsp(P_2.r_4, in(P_2.enter\_cs)) = in(P_2.wait\_for)$
$wcr(P_2.r_4, in(P_2.enter\_cs)) = in(P_1.flag_1 = 0) \vee in(P_1.turn = 1)$
end of $P_2.r_4$;

$P_2.r_5$ :: % Release critical section %
$wsp(P_2.r_5, in(P_2.flag_2 = 0)) = in(P_2.enter\_cs)$
$wcr(P_2.r_5, in(P_2.flag_2 = 0)) = true$
end of $P_2.r_5$;

$P_2.r_6$ :: % Execute non critical section %
$wsp(P_2.r_6, in(P_2.ncs)) = in(P_2.flag_2 = 0)$
$wcr(P_2.r_6, in(P_2.ncs)) = true$
end of $P_2.r_6$;

6 Proof of Correctness

6.1 Mutual Exclusion

Assertion. *The number of processes executing in the critical section is never more than* 1.

Proof. Assume the contrary, that is both processes P_1 and P_2 are in their critical sections. Therefore, weakest preconditions for transition rules $P_1.r_4$ and $P_2.r_4$ hold together. Thus the following condition is true.

$$\begin{aligned}&in(P_1.wait\_for) \wedge \{in(P_2.flag_2 = 0) \vee\\ &in(P_2.turn = 2)\} \wedge in(P_2.wait\_for) \wedge\\ &\{in(P_1.flag_1 = 0) \vee in(P_1.turn = 1)\}\\ &\Rightarrow in(P_1.turn = 1) \wedge in(P_2.turn = 2) \Rightarrow false.\end{aligned}$$

Because *turn* is a variable, it can hold at most one value at a time assigned by either process. Therefore, the assertion must be true and mutual exclusion is ensured. □

6.2 Liveness

Assertion. *A process that wishes to enter the critical section eventually does so.*

Proof. We prove it by contradiction. Assume that process P_1 is in non-critical section, process P_2 is trying to enter its critical section and is blocked. Therefore, weakest preconditions for transition rules $P_1.r_6$ and $P_2.r_3$ hold together. Thus the following condition is true.

$$\begin{aligned}&in(P_1.flag_1 = 0) \wedge \{in(P_2.turn = 2) \wedge\\ &in(P_1.flag_1 = 1)\}\\ &\Rightarrow in(P_1.flag_1 = 0) \wedge in(P_1.flag_1 = 1) \Rightarrow false.\end{aligned}$$

Because $flag_1$ is a variable local to process P_1, it can hold at most one value at a time assigned by process P_1. Therefore, the correctness of the assertion is well guarded. □

6.3 Deadlock Freeness

Assertion. *Both processes are never blocked together.*

Proof. Assume the contrary. Process P_1 attempting to enter its critical section, process P_2 attempting to enter its critical section, and both are blocked in the wait state. Therefore, weakest preconditions for transition rules $P_1.r_3$ and $P_2.r_3$ hold together. Thus the following condition is true.

$$\begin{aligned}&\{in(P_1.turn = 1) \wedge in(P_2.flag_2 = 1)\} \wedge\\ &\{in(P_2.turn = 2) \wedge in(P_1.flag_1 = 1)\}\\ &\Rightarrow in(P_1.turn = 1) \wedge in(P_2.turn = 2) \Rightarrow false.\end{aligned}$$

Because *turn* is a variable, it can hold at most one value at a time assigned by either process. Therefore, the assertion must be true and deadlock is not possible in the system. □

6.4 Bounded Waiting

If a process of the system is executing its critical section repeatedly, it is said that the process has locked out the shared resource. Now, if the other process of the system is also trying to enter its critical section, it has to wait indefinitely. Therefore, the second process is starving. This situation must not arise. There should be bounded the waiting time for the second process to enter its critical sections. Thus the algorithm should be lockout-free.

Assertion. *A single process is never allowed to execute its critical section repeatedly, while the other process is also waiting to enter its critical section.*

Proof. Assume the contrary, that is process P_1 is starving and process P_2 is executing its critical section repeatedly. Let us consider the situation when process P_1 is in wait state and process P_2 has just released critical section. When process P_1 is in wait state, we have the following condition.

$$\begin{aligned}&\{in(P_1.wait\_for) \wedge in(P_1.turn = 1) \wedge\\ &in(P_2.flag_2 = 1) \wedge \{in(P_1.flag_1 = 1)\}.\end{aligned} \quad (1)$$

Now, if process P_2 comes out of the critical section, the post condition of the transition rule $P_2.r_5$, will become true. That is, the component $in(P_2.flag_2 = 1)$, of (1), will change to the new value $in(P_2.flag_2 = 0)$. Thus, the condition, given above in (1), gets modified as follows.

$$\begin{aligned}&\{in(P_1.wait\_for) \wedge in(P_1.turn = 1) \wedge\\ &in(P_2.flag_2 = 0) \wedge \{in(P_1.flag_1 = 1)\}.\end{aligned} \quad (2)$$

Hence, the following condition, given in (3), would also hold being weaker than the condition given above, in (2).

$$\{in(P_1.wait\_for) \wedge in(P_2.flag_2 = 0)\} = turn. \quad (3)$$

Since we also observe that

$$\begin{aligned}&\{in(P_1.wait\_for) \wedge in(P_2.flag_2 = 0)\}\\ &\Rightarrow wp(P_1.r_4, in(P_1.enter\_cs)).\end{aligned} \quad (4)$$

Therefore, process P_1 would enter the critical section before process P_2. This is so because the post condition, of the transition rule $P_2.r_2$, makes $in(P_2.turn = 2) = true$. This condition, along with the condition $in(P_1.flag_1 = 1)$ which was obtained as the post condition of the transition rule $P_1.r_1$, would make

$$wp(P_2.r_3, in(P_2.wait\_for)) = true. \qquad (5)$$

It is obvious from the above two equations (4) and (5), process P_2 will not be allowed to enter critical section before process P_1 does so. Therefore, the initial assumption about the system is wrong and the starvation freedom is also guaranteed.

7 Conclusion

The purpose of this article has been to show how successfully one can use split precondition logic as formal means to appropriately model and prove the validness of distributed algorithms. In the case study of Peterson's two-process algorithm we have formally proved safety and liveness properties. While reasoning, using the split precondition logic, we reasoned backward to establish that certain states could not have been reached. The backward reasoning reduces the search space. Therefore, it is preferable being goal oriented[36]. The approach, presented in this paper, suggests how split precondition logic can be used to handle a class of distributed algorithms in which processes communicate through shared variables. Though the strength of our modeling technique is simplicity, accuracy has not been compromised for the sake of simplicity. Nevertheless, our approach needs careful human effort. However, in our logic, the correctness is ensured by proving assertions that are formulas in predicate logic. These formulas must be embedded into the system during its design phase. Dijkstra[35] also mentioned this in connection with the loop invariants. The proof of these formulas requires standard predicate logic rules and also the transition rules of the system in question. Since we propose to specify a system by its transition rules, this formalization is available to us. It should therefore be possible to develop a rule-based system to evaluate the correctness of the assertions. One can also think of a proof system that may be intelligent enough to consult with the user and update its rule base.

References

[1] Ammons G, Bodik R, Larus J R. Mining specifications. In *Proc. 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* Portland, Oregon, January 16–18, 2002, pp.4–16.

[2] Cook J E, Wolf A L. Event-based detection of concurrency. In *Proc. ACM SIGSOFT'98 Symposium on the Foundation of Software Engineering,* Orlando, FL, November 1998, pp.35–45.

[3] Raz O, Koopman P, Shaw M. Semantic anomaly detection in online data sources. In *Proc. ICSE'02 24th International Conference on Software Engineering,* Orlando, Florida, May 22–24, 2002.

[4] Hangal S, Lam M S. Tracking down software bugs using automatic anomaly detection. In *Proc. ICSE'02 24th International Conference on Software Engineering,* Orlando, Florida, May 22–24, 2002.

[5] Win T N, Ernst M. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report, MIT-LCS-TR-841, MIT Lab for Computer Science, 200 Technology Square, Cambridge, MA, USA, May 2002.

[6] Agha G. Actors: A model of concurrent computation. In Distributed Systems, MIT Press, 1986.

[7] Hoare C A R, Jifeng H E. The prespecification. *Information Processing Letters,* 1987, 24: 127–132.

[8] Hoare C A R. Communicating Sequential Processes. Prentice Hall, 1985.

[9] Chauchen Z, Ravn A P, Hoare C A R. A calculus of durations. *Information Processing Letters,* 1991, 40(5): 269–276.

[10] Hennessy M, Milner R. Algebraic laws for nondeterminism and concurrency. *JACM,* 1985, 32(1): 137–161.

[11] Holmstrom S. Hennessy-Milner logic with recursion as a specification language, and a refinement calculus based on it. In Specification and Verification of Concurrent Systems, Springer-Verlag, 1990, pp.294–330.

[12] Lamport L. Specifying concurrent program modules. *ACM Trans. Programming Languages and Systems,* 1982, 5(2): 190–222.

[13] Lamport L. The temporal logic of actions. *ACM Trans. Programming Languages and Systems,* May 1994, 16(3): 872–923.

[14] Spivey J M. Understanding Z: A Specification Language and Its Formal Semantics. Cambridge University Press, 1988.

[15] Milner R. A calculus of communicating systems. *LNCS 92*, Springer-Verlag, 1980.

[16] Nelson G. A generalization of Dijkstra's calculus. *ACM Trans. Programming Languages and Systems,* 1989, 11(4): 517–561.

[17] Apt K R, Francez N, deRover W P. A proof system for communicating sequential processes. *ACM Trans. Programming Languages and Systems,* 1980, 2(3): 359–385.

[18] Owicki S, Gries D. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM,* May 1976, 19(5): 279–284.

[19] Weber R. Where can I get gas round here? — An application of a design methodology for distributed systems. *LNCS 490*, Springer-Verlag, 1991, pp.143–166.

[20] Pau I, Rents I, Schreiner W. Verifying mutual exclusion and liveness properties with TLA. Technical Report 00-06, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, January 2000.

[21] Peterson G L. Myths about the mutual exclusion problem. *Information Processing Letters,* June 1981, 12(3): 115–116.

[22] Alpern B, Schneider F B. Verifying temporal properties without temporal logic. *ACM Trans. Programming Languages,* January 1989, 11(1): 147–167.

[23] Eilenberg S. Automata, Languages and Machines. Vol A, Academic Press, New York, 1974.

[24] Shankar A U. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys,* September 1993, 25(3): 225–262.

[25] Manna Z, Pnueli A. Temporal Verification of Reactive Systems: Safety. Springer-Verlag, New York, 1995.

[26] Manduchi G, Moro M. Automatic verification for a class of distributed systems. *Distributed Computing,* 2000, 13(3): 127–143.

[27] Finney K. Mathematical notation in formal specification — Too difficult for the masses? *IEEE Trans. Software Engineering,* February 1996, 22(2): 158–159.

[28] Lynch N A, Tuttle M R. An introduction to input/output automata. *CWI-Quarterly,* September 1989, 2(3): 219–246.

[29] Lynch N A. Distributed Algorithms. Morgan Kaufmann, San Francisco, CA, 1996.

[30] Garland S J, Lynch N A, Vaziri M. IOA: A language for specifying, programming, and validating distributed systems. Technical Report, MIT Laboratory for Computer Science, 1997.

[31] Garland S J, Guttag J V. A guide to LP: The larch prover. Technical Report 82, Digital Equipment Corporation, Systems Research Center, 31 December 1991.

[32] Ernst M D, Cockrell J, Griswold W G, Notkin D. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Engineering,* February 2001, 27(2): 1–25.

[33] Chandi K M, Sanders B A. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming,* 1995, 24: 129–148.

[34] Singh A K, Bandyopadhyay A K. An algorithm to find the condition for deadlock in a distributed system. In *Proc. WDC'2000,* 2000, pp.70–73.

[35] Dijkstra E W. A Discipline of Programming. Prentice Hall, 1976.

[36] Ben-Ari M. Mathematical Logic for Computer Science. Prentice Hall, 1993.

Awadhesh Kumar Singh received the B.E. degree in computer science & engineering from Gorakhpur University, Gorakhpur, India in 1988. He received the M.E. and Ph.D. (Engg) degrees in the same area from Jadavpur University, Kolkata, India. He is a faculty member in Computer Engineering Department, National Institute of Technology, Kurukshetra, India. His present research interest is distributed systems.

Anup Kumar Bandyopadhyay received the B.E. (Tel.E.), M.E. (Tel.E.), and Ph.D. (Engg) degrees from Jadavpur University, Calcutta, India in 1968, 1970 and 1983, respectively. From 1970 to 1972 he worked with the Microwave Antenna System Engineering Group of the Indian Space Research Organization. In 1972 he joined the Department of Electronics and Telecommunication Engineering, Jadavpur University, where he is currently a professor. His research interests include computer communication networks and distributed systems.