Reasoning about Protocols using Dijkstra' s Calculus

Awadhesh Kumar Singh
Department of Computer Engineering, National Institute of Technology,
Kurukshetra, India 136119
Email: aksinreck@rediffmail.com
and
Anup Kumar Bandyopadhyay
Department of Electronics and Telecommunication Engineering,
Jadavpur University, Kolkata, India 700032
Email: anupbandyopadhyay@hotmail.com

ABSTRACT

A mathematical model for the specification and verification of a data link layer protocol is proposed. The weakest precondition calculus, developed by Dijkstra, originally for sequential programs, has been chosen for this purpose. It is demonstrated that the *wp*–calculus provides a basis, not only for the modeling but also, for a straightforward and thorough analysis of large and complex distributed systems like data link layer protocol. This analysis contributes to the understanding of the system and could lead to an improvement in the design. The technique has been illustrated by describing the sliding window protocol.

Keywords: weakest precondition, guarded process, non-deterministic selection, protocol, weakest cooperation, correctness

1. INTRODUCTION

Large number of formal methods exists to capture various types of systems. Hall in his Seven Myths[1] asserted that mathematics for specification should be easy. People find formal specifications difficult to read because of the large use of the symbols. The notational difficulties are more in writing specifications, due to the need for great attention to detail and correct use of mathematical statements. The weakest precondition (*wp*) calculus[2] uses first order predicate logic, which is very easy to apply. Moreover, through *wp* approach one can derive cause from effect, that is – precondition from post conditions. Therefore, it is much more sound[3]. Dijkstra developed it originally for reasoning about the correctness of sequential programs. We propose to extend the proof technology into the realm of modeling and reasoning about the correctness of distributed systems, in order that it can be applied to successfully verify either a component of a distributed system or an entire distributed system.

Distributed system is a collection of processor–memory pairs connected by a local area network or distributed over a large geographical area. The processors communicate in various unpredictable ways, because distributed systems are inherently concurrent, asynchronous and non-deterministic. These characteristics make them more complex than sequential systems [4]. Normally system specification depicts operational requirements and it does not include the properties like liveness, fairness, deadlock freedom, mutual exclusion, etc. Implementer' s goal should be to include these properties in order to achieve the correct system design. The design of distributed systems is known to be a complex task, because the behavior of a distributed system results from interactions between concurrent processes of which the system consists [5].

The classical example of distributed system is a well-known prototypical resource allocation system involving allocation of pair wise shared resources in a ring of processors, that is – dining philosophers system. Such systems can be successfully specified and verified using our proof technology [6]. However, in this type of system there is possibility of occurrence of only deadlock problem, hence the formal verification is easier. The distributed algorithms constitute a more complex class of distributed systems. Even though the code for a distributed algorithm may be short, the fact that many processors execute the code in parallel, with steps interleaved in some undetermined way, implies that there are many different ways in which the algorithm can behave, even for the same inputs. It is generally impossible to understand such algorithm by predicting exactly how it will execute. As many processes concurrently behave and timing conditions are strict, it is more difficult to verify them by any formal methodology. The processes of such system communicate through message transaction. In the centralized environment, the message transmitted by a process is almost guaranteed to reach at the site waiting for the message. The same is not always true in distributed environment. This fact complicates the system furthermore. Newmann [7] wrote, "Distributed systems have distributed risks". The risks tend to increase and become more insidious as ' distributivity' increases. Formal techniques can be used to reduce risks [8]. Here, we will use our technique for the same purpose. Let us consider a system with more ' distributivity' where a message generated from a particular site is not guaranteed to reach the destination. We will investigate how our approach can be used to model and verify such systems.

In order to ensure the correctness of a typical distributed system, in addition to deadlock freedom, the properties, like safety, liveness, and livelock freedom also, must be verified. All communication protocols do have these properties. We consider the sliding window protocol, one of the popular data link layer protocols. Since, this example presents a situation where all of the above enumerated properties can be verified, therefore, it lends itself as the most general and perfect candidate, not a mere case study, to test and manifest the power of any formal verification technique.

Pnueli and Harel introduced the concept of a *reactive system*, a system whose behavior is characterized by non-termination and on-going interaction with an environment over which the system has little control [9]. Network protocols can be modeled as reactive systems [10]. The message transfer in sliding window protocol is

through a channel. In order to simplify the problem, one may not assume the existence of the channel or one may assume the channel is co-operative one. But *wp* paradigm will not be of much interest when applied to this *open system*, which interacts with an environment. Thus one should assume a hostile environment [11]. Therefore, we assume the channel is noisy one. In order to specify sliding window protocol through weakest precondition logic, firstly we have modeled a noisy channel via this logic. The predicates to ensure correctness have been found on the basis of working of the protocol. Modeling is parallel to the communicating processes.

2. OVERVIEW OF THE *WP*–CALCULUS

The specification language *wp* has – like every formal language – a well-defined syntax and semantics. An informal description based on a formal mathematical model is being given in the following paragraphs.

A set *P.X* of states and a set *P.R* of state transition rules define a process *P*. On the similar lines a set *S.P* of processes interacting through message transactions define a system *S*. The predicate 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 $initial(P.x_0)$. 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 other. For firing any transition rule *P.r* there exist a corresponding weakest precondition *wp*(*P.r*, *Q*). If the system state satisfies the condition *wp*(*P.r*, *Q*) then execution of the transition rule *P.r* will eventually establish the post condition *Q*. This condition can be divided into two parts[12].

(i) *wsp*(*P.r*, *Q*), termed as weakest self pre-condition and is related to the process *P* itself.
(ii) *wcr*(*P.r*, *Q*), termed as weakest co-operation requirements and includes the co-operation requirements from other processes.

Thus, the total weakest precondition will be given by

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

Since, the co-operation requirements have already been included in the weakest precondition in this approach, separate proof of co-operation is not necessary here which was required in Chandi–Sanders approach[13]. Any transition rule *P.r* is described jointly by the weakest pre-condition *wp*(*P.r*, *Q*) and the post condition *Q*.

A non-deterministic state transition rule *P.r* may include number of different sub-rules each of which requires a definite pre-condition, called guard, to be satisfied for its execution. Execution of a sub-rule will change the state of the process *P* as well as the state of one of the co-operating processes whose active co-operation is necessary for this execution. If the pre-condition for more than one sub-rules are satisfied then one of them is selected, non-deterministically, for execution. Let there be *n* number of sub-rules denoted by $P.r^i : i = 1,.,. n$. On 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 $s_i : i = 1,.,. m$. 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 denotes 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 ::

$Q \equiv \exists\, i : Q_i$;

$wp(P.r, Q) = (\exists\, k : B_k)$

$\wedge\, (\forall\, i \bullet B_i \Rightarrow wr(select, s_i))$

$\wedge\, (\forall\, j \bullet s_j \Rightarrow wp(P.r^j, Q_j))$;

end of the transition rule *P.r*;

The operational model can be described by state transition rules. These rules are completely defined by their weakest pre-condition pairs. However, operational specification may not be sufficient to include all the system requirements. One must also specify system properties that should be maintained before and after each state transition. Best way to do this is to construct a condition *Q* related to the system states and the property requirements, such that

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

In other words a system invariant must also be specified. For a guarded command the system invariant Q must follow a similar condition that is given by

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

3. SPECIFICATION OF A NOISY CHANNEL

As given in [14], we consider a noisy channel that accepts messages for transmission. Depending on the noise condition during the transmission time, the message is presented at the output or it may be lost. We describe this channel as follows. We define the following states for the channel.

	STATE	SEMANTICS
1.	Channel_idle	Channel is idle
2.	Channel_launched	A frame has been launched on the Channel
3.	Channel_produced	A frame is presented to the receiver

We can now describe the behavior of the channel as follows.

Process Channel ();
States : channel_idle, channel_launched, channel_produced;
Initial condition : in(channel_idle);
Transition rules
Channel . r_1
def Q=in(channel_launched);
wsp(Channel . r_1, Q)=in(channel_idle);
wcr(Channel . r_1, Q)=in(Ps . sent);
end of the transition rule Channel . r_1;
Channel . r_2
def $Q=Q^1 \vee Q^2$
Q^1=in(channel_produced);
Q^2=in(channel_idle)
(in(channel_launched) $\Rightarrow$ wp(Select, S^1))
$\wedge$ (in(channel_launched) $\Rightarrow$ wp(Select, S^2))
$\wedge$ ($S^1 \Rightarrow$ wp(Channel . r_2^1, Q^1))
$\wedge$ ($S^2 \Rightarrow$ wp(Channel . r_2^2, Q^2))
end of the transition rule Channel . r_2;

end of transition rules;
We assume a sender process Ps is running concurrently and moves to the state Ps . sent when it presents a message at the channel input. There exists another transition rule for the channel. When the receiver process receives the message the channel invokes another transition rule and moves to the idle state.

4. FORMALIZATION OF THE SLIDING WINDOW PROTOCOL

4.1 Informal Specification

In sliding window protocol[15], each outbound frame contains a sequence number, ranging from 0 up to some maximum. At any instant of time, the sender maintains a fixed size buffer corresponding to set of frames, sent but yet not acknowledged, with their sequence numbers. This buffer is termed sender's window. This storage is done for possible retransmission; since sent frames may ultimately be lost or damaged in transit. Since it has multiple outstanding frames, it maintains multiple logical timers, one per outstanding frame. Each frame times out independently of all the other once. Similarly the receiver also maintains a receiver's window corresponding to the set of frames it is permitted to accept. The receiver has a buffer reserved for each sequence number within its window. Associated with each buffer is a bit telling whether the buffer is full or empty? Whenever a frame arrives, its sequence number is checked to see if it falls within the window. If so, and if it has not already been received, it is accepted and stored. An acknowledgement is sent also, if its predecessor frame has been acknowledged. Whenever the receiver has reason to suspect that an error has occurred, it sends a negative acknowledgement (NAK) frame back to the sender. Such a frame is a request for retransmission of the frame specified in the NAK. The Sender retransmits a frame, either on receiving NAK or on being timed out, whichever is earlier. Any frame falling outside the window is discarded without comment. The sender's window and receiver's window need not have the same lower and upper limits, or even have the same size. A frame buffer is released if buffer for its predecessor frame has already been released. Sender releases buffer after receiving acknowledgement while receiver does the same after sending acknowledgement. We define the states for different processes of the system as follows.

4.2 Formal Specification

Like in other modeling techniques, we also make certain assumptions, which provide framework for analysis of the system under consideration. 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 order of interleaved execution. Almost every model used for correctness analysis assumes that the execution of a concurrent system can be viewed in terms of events that can be considered atomic [16]. Our technique also views the execution of the system in terms of the atomic events. These events are communication with the other process, that is, a message transfer. Due to our assumption regarding atomicity, we can formalize a data link layer protocol as a state transition system. We have selected the sliding window protocol for this purpose. Now, we model the system by considering the presence of a set of processes in the system, namely
1. sender process Ps, which sends frames in to the channel.
2. receiver process Pr, which receives frames from the channel.
3. Frame_OK is a flag, which shows that incoming frame does not contain any error.
4. Timer process
5. Win_rec is also a flag showing that the receiver process has received all frames of the current window.
6. producer process Pp, which delivers frames to the sender process.

4.2.1 States for the sender process Ps

	STATE	SEMANTICS
1.	Ps . rts_i	Ps is ready to send frame_i
2.	Ps . sent_i	Ps has sent frame_i
3.	Ps . rec_ack_i	Ps has received acknowledgement for frame_i
4.	Ps . repeat_i	Ps resends frame_i and restarts its local Timer
5.	Ps . rec_nak_i	Ps has received negative acknowledgement for frame_i
6.	Ps . release_buf_i	Ps has released buffer occupied by frame_i

4.2.2 States for the receiver process Pr

	STATE	SEMANTICS
1.	Pr . rtr_i	Pr is ready to receive frame_i
2.	Pr . rec_i	Pr has received frame_i
3.	Pr . sent_ack_i	Pr has sent acknowledgement for frame_i
4.	Pr . sent_nak_i	Pr has sent negative acknowledgement for frame_i
5.	Pr . inbuf_i	Pr has put frame_i in buffer
6.	Pr . release_buf_i	Pr has released buffer occupied by frame_i
7.	Pr . no_accept_i	Pr will not accept frame i

4.2.3 States for Frame_OK flag

	STATE	SEMANTICS
1.	Frame_OK=true	If the frame arrived at the receiver does not have any error
2.	Frame_OK=false	If the frame arrived at the receiver has some error

4.2.4 States for Timer process

	STATE	SEMANTICS
1.	Timer_idle_i	Timer for frame_i is idle
2.	Timer_start_i	Timer for frame_i starts
3.	Timer_end_i	Timer for frame_i stops
4.	Timer_restart_i	Timer for frame_i restarts

4.2.5 States for Win_rec flag

	STATE	SEMANTICS
1.	Win_rec=false	All frames related to current window have not been received, except frame i
2.	Win_rec=true	All frames related to current window have been received, except frame i

4.2.6 States for the producer process Pp

	STATE	SEMANTICS
1.	Pp . idle	Producer is idle
2.	Pp . produced_i	Producer has delivered frame i to sender process

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

4.2.7 Process Ps: identified by Ps; initialized as in(Ps . rts_i)

Transition rule

Let,

Ws = size of sender' s window

i = frame in sender' s window where i will assume values 0, 1, . .,.(Ws – 1) in sequence

j = next frame in sender' s window such that j = (i + 1) mod Ws

k = arbitrary frame in sender' s window

i' = first frame in sender' s window after sliding such that i' = k + 1

The range of values assumed by i' is same as i.

Ps . send_frame

def Q1=in(Ps . rts_j) ∧ in(Ps . sent_i) ∧ in(Timer_start_i) ∧ in(Channel_launched)

Q2=in(Ps . release_buf_i) ∧ in(Timer_end_i) ∧ in(Ps . rts_j) ∧ in(Ps . rec_ack_i)

Q3=in(Ps . repeat_i) ∧ in(Timer_restart_i) ∧ in(Ps . rts_k) ∧ in(Ps . rec_nak_i)

Q4=in(Ps . repeat_i) ∧ in(Timer_restart_i) ∧ in(Ps . rts_k)

Q5=in(Ps . release_buf_i-to-k) ∧ in(Ps . rts_i')

Q=Q1 ∨ Q2 ∨ Q3 ∨ Q4 ∨ Q5

B1=in(Ps . rts_i) ∧ in(Pp . produced_i)

B2=in(Ps . sent_i) ∧ in(Timer_start_i) ∧ in(Channel_produce_ack_i)

B3=in(Ps . rts_k) ∧ in(Channel_produce_nak_i)

B4=in(Ps . rts_k) ∧ in(Timer_end_i)

B5=in(Ps . repeat_i) ∧ in(Channel_produce_ack_k)

wp(Ps . send_frame, Q) =(B1 ∨ B2 ∨ B3 ∨ B4 ∨ B5)
∧ (B1⇒wr(select, in(Ps . send_frame . s1)))
∧ (B2⇒wr(select, in(Ps . send_frame . s2)))
∧ (B3⇒wr(select, in(Ps . send_frame . s3)))
∧ (B4⇒wr(select, in(Ps . send_frame . s4)))
∧ (B5⇒wr(select, in(Ps . send_frame . s5)))
∧ (in(Ps . send_frame . s1)⇒wp(Ps . send_frame^1, Q1))
∧ (in(Ps . send_frame . s2)⇒wp(Ps . send_frame^2, Q2))
∧ (in(Ps . send_frame . s3)⇒wp(Ps . send_frame^3, Q3))
∧ (in(Ps . send_frame . s4)⇒wp(Ps . send_frame^4, Q4))
∧ (in(Ps . send_frame . s5)⇒wp(Ps . send_frame^5, Q5))

End of the transition rule Ps . send_frame ;

End of the process Ps ;

4.2.8 Process Pr: identified by Pr; initialized as in(Pr . rtr_i)

Transition rule

Let,

Wr = size of receiver' s window

i = frame in receiver' s window where i will assume values 0, 1, . .,.(Wr – 1) in sequence

j = next frame in receiver' s window such that j = (i + 1) mod Wr

k = arbitrary frame in receiver' s window

i' = first frame in receiver' s window after sliding such that i' = k + 1

The range of values assumed by i' is same as i.

Pr . receive_frame

def Q1= in(Pr . rec_i) ∧ in(Pr . rtr_j) ∧ in(Pr . sent_ack_i) ∧ in(Pr . release_buf_i) ∧ in(Win_rec=false)

Q2=in(Pr . rtr_i) ∧ in(Pr . inbuf_k)

Q3= in(Pr . rec_i) ∧ in(Pr . rtr_i') ∧ in(Pr . sent_ack_k) ∧ in(Pr . release_buf_i-to-k) ∧ in(Win_rec=true)

Q4=in(Pr . rtr_i) ∧ in(Pr . sent_nak_i)

Q5=in(Pr . no_accept_k)

Q=Q1 ∨ Q2 ∨ Q3 ∨ Q4 ∨ Q5

B1=in(Pr . rtr_i) ∧ in(Channel_produce_i) ∧ in(Frame_i_OK=true)

B2=in(Pr . rtr_i) ∧ in(Channel_produce_k) ∧ in(Frame_k_OK=true) ∧ ¬in(Pr . inbuf_k)

B3=in(Pr . rtr_i) ∧ in(Channel_produce_i) ∧ in(Pr . inbuf_j-to-k) ∧ in(Frame_i_OK=true)

B4=in(Pr . rtr_i) ∧ in(Channel_produce_i) ∧ in(Frame_i_OK=false)

B5=in(Pr . inbuf_k) ∧ in(Channel_produce_k)

wp(Pr . receive_frame, Q) =(B1 ∨ B2 ∨ B3 ∨ B4 ∨ B5)
∧ (B1⇒wr(select, in(Pr . receive_frame . s1)))
∧ (B2⇒wr(select, in(Pr . receive_frame . s2)))
∧ (B3⇒wr(select, in(Pr . receive_frame . s3)))
∧ (B4⇒wr(select, in(Pr . receive_frame . s4)))
∧ (B5⇒wr(select, in(Pr . receive_frame . s5)))
∧ {(in(Pr . receive_frame . s1)⇒ wp(Pr . receive_frame^1, Q1))}
∧ {(in(Pr . receive_frame . s2)⇒ wp(Pr . receive_frame^2, Q2))}
∧ {(in(Pr . receive_frame . s3)⇒ wp(Pr . receive_frame^3, Q3))}
∧ {(in(Pr . receive_frame . s4)⇒ wp(Pr . receive_frame^4, Q4))}
∧ {(in(Pr . receive_frame . s5)⇒ wp(Pr . receive_frame^5, Q5))}

End of the transition rule Pr . receive_frame ;

End of the process Pr ;

4.3 PROOF OF CORRECTNESS

We define an operator "leads to" (symbolized as "⟶") in *wp* environment with the following semantics.

B ⟶ Q implies ∃r : wp(r, Q) = B (1)

where B and Q are guards and post-conditions respectively corresponding to transition rule r. In order to function properly the protocol must satisfy both, the safety and liveness properties.

4.3.1 Safety property

All frames must be received without repetition. We will prove it in two parts.

(a) Transition rule B5 ⟶ Q5 of receiver process reveals that weakest precondition for not accepted frame k is

"frame k is already in buffer of receiver". Thus any frame once received will never be received again.

(b) 'All' frames must be received. We interpret, "if all frames of current window are received, only then receiver should become ready to receive first frame of next window". This condition can be represented in predicate form as follows.

$$in(Pr . rec\_i) \wedge in(Win\_rec=true) \longrightarrow in(Pr . rtr\_m) \quad (2)$$

As defined previously m= i', where i' is sequence number of first frame in next window.

Transition rule B3 $\longrightarrow$ Q3 exhibits that

$$B3 \longrightarrow \{in(Pr . rec\_i) \wedge in(Win\_rec=true) \wedge in(Pr . rtr\_m) \wedge I\} \quad (3)$$

Where I= in(Pr . sent_ack_k) $\wedge$ in(Pr . release_buf_i-to-k)

Also the predicate shown in Eq. (2) is weaker than that of in Eq. (3). Thus correctness of the predicate shown in Eq. (2) is ensured.

4.3.2 Liveness property

All frames must be transmitted with finite delay.

Transition rule B1 $\longrightarrow$ Q1 of the sender process reveals that transition from ready to send frame i to ready to send frame j (where j=i+1) needs no co-operation from any other process except it's producer process. Producer process supplies frames from upper layer to sender process. Thus, every frame supplied by producer process will eventually be transmitted.

4.3.3 Deadlock freedom

We observe, mainly sender process ensures liveness and receiver process ensures safety. Sender and receiver are two non-competing processes, represented through guarded commands. In guarded commands, though more than one guards can be true at a time, only one true guard will be selected and corresponding statement will be executed. Thus deadlock freedom is ensured.

4.3.4 Livelock freedom

Assume that acknowledgement of some frame i is lost in the channel. Subsequently, timer for frame i will expire, sender will transmit frame i again and restart it's timer. Receiver sends acknowledgement for any particular frame at most once. Thus, sender will never receive the acknowledgement for frame i and will be livelocked in transmitting frame i.

In order to prevent this situation, while repeatedly sending a particular frame, say i, if channel produces acknowledgement for a frame with higher sequence number, say k, then buffer for all frames having sequence numbers less than or equal to k are released, that is, window slides. In other words, sender becomes ready to send first frame of next window. We interpret, "if sender transmits a frame then corresponding buffer would eventually be released irrespective of acknowledgement reception. Following predicate can represent this.

$$\{in(Ps . repeat\_i) \wedge in(Channel\_produce\_ack\_k)\} \longrightarrow \{in(Ps . release\_buf\_i\text{-}to\text{-}k) \wedge in(Ps . rts\_m)\}$$

Recall that m= i', where i' is sequence number of first frame in next window.

This is fully guarded by B5 $\longrightarrow$ Q5 of sender process.

Thus, live-lock condition will never occur.

5. COMPARISON WITH TLA APPROACH

One of the popular techniques for verifying systems is Temporal Logic of Actions abbreviated as TLA. There all specifications, including the requirement, have the form of a state machine [17] and the specified actions describe the transition between the states. In order to distinguish between the value before and after the action was executed, the prime operator is introduced. The definition of action describes the behavior of distinct variables that change values as well as the remaining free variables that are left unchanged. UNCHANGED statements for the remaining variables have supplemented the definition. Similarly additional action operators are provided which allow to state whether an action affects a variable. What remains unchanged is stated in TLA action definition; and how the control state changes is described in the TLA formula [18].

#1 Any action can only be executed if it is enabled. Thus, along with action definitions enabling conditions are also taken care for all actions, deterministic and non-deterministic. In present approach using guarded command, one guard has to be made true before the corresponding transition rule is executed. However, it is not required in deterministic type of actions. Hence specification is shorter than TLA formula and proofs of various properties are less complicated.

#2 TLA provides a clear separation between safety and liveness properties. The specifications in [19, 20] concern only safety properties. It involves very long and tiresome proof. In order to obtain complete specifications it will require another effort, to the tune of similar size and complexity, in order to prove liveness properties.

#3 In order to introduce full de-coupling between sender and receiver, [19, 20] requires two communication channels, which is reduced to one in our work using weakest pre-condition modeling approach.

#4 Sliding Window re-transmission mechanism has been modeled with linear arithmetic in [19, 20]. Modular arithmetic has the advantage of partitioning the long proof in to smaller ones. At the same time it introduces an overhead also. One has to include a series of proofs exhibiting the fact that each module implements its predecessor, who is not needed in linear arithmetic, used by us.

#5 Getting the invariant of implementation that is, lower level specification correctly implements a higher-level one [21], used for the refinement mapping proof, is also tedious. There is no systematic method to invent the suited invariant for the refinement proof. It relies on the experience gained from a deep study of the problem in question [22]. Moreover these refinement proofs are not simple. The proof is straightforward only when mapping does not introduce stuttering steps. Similar overheads are absent in out approach.

6. CONCLUSION

We have shown how a distributed system may be represented by using the notation of the weakest pre-condition calculus through the example of sliding window protocol. Any distributed system consists of a collection of communicating processes, and these processes can be modeled in this notation. Thus any distributed system may be modeled in the notation. The goal was to formalize and prove its correctness. As the example, considered here, includes almost all desirable properties of a typical distributed system, it proves, without loss of generality, the effectiveness of our proof technology to verify the correctness of any distributed system. Although we have used first order logic, accuracy has not been compromised

for the sake of simplicity. It also demonstrates the suitability of the weakest precondition calculus for evaluating the correctness of the class of distributed systems that communicate through message transaction.

A preliminary version of this work has appeared in [23]. The present exposition is an extension of our earlier work and is more exhaustive, free from some anomalies pointed out during presentation of the preliminary version. The proofs of properties are simpler and easier to understand.

7. REFERENCES

[1] J. A. Hall, Seven myths of formal methods, IEEE Software, vol. 7, September 1990, pp. 11–19.

[2] E. W. Dijkstra, A discipline of programming, Prentice Hall, 1976.

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

[4] David S. Rosenblum, Specifying concurrent systems with TSL, IEEE Software, 8(3), May 1991, pp. 52-61.

[5] Yoshinao Isobe and Kazuhito Ohmaki, A process logic for distributed system synthesis, Proc. APSEC 2000, IEEE 7th Asia–Pacific Software Engineering Conference, 2000, pp. 62-69.

[6] A. K. Singh, Ph. D. Thesis, Faculty of Engineering and Technology, Jadavpur University, Kolkata, India, May 2003.

[7] Peter G. Neumann, Distributed systems have distributed risks, Communications of the ACM, 39(11), Nov 1996, Page 130.

[8] Peter G. Neumann, Using formal methods to reduce risks, Communications of the ACM, 39(7), Jul 1996, Page 114.

[9] D. Harel and A. Pnuali, On the development of reactive systems, In K. R. Apt, editor, Logics and Models of Concurrent Systems, vol. F-13 of NATO ASI Series, Springer-Verlag, 1985, pp. 477-498.

[10] Panagiotis Manolios and Richard Trefler, Safety and liveness in branching time, Proc. IEEE 16th Annual Symposium on Logic in Computer Science, 2001, pp. 366-374.

[11] Orna Kupferman and Moshe Y. Vardi, Synthesizing distributed systems, Proc. IEEE 16th Annual Symposium on Logic in Computer Science, 2001, pp. 389-398.

[12] A. K. Bandyopadhyay and J Bandyopadhyay, On the derivation of a correct deadlock free communication kernel for loop connected message passing architecture from its user's specification, Journal of System Architecture, vol. 46, No. 13, 2000, pp. 1257–1261.

[13] K. M. Chandi and B. A. Sanders, Predicate transformers for reasoning about concurrent computation, Science of Computer Programming, vol. 24, 1995, pp. 129–148.

[14] A. K. Bandyopadhyay, A mathematical model for the specification of data link layer protocols, In Proc. COMNAM-2000, December, 2000, pp. 109–114.

[15] A. S. Tanenbaum, Computer Networks 3/e, Prentice Hall of India, 1998.

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

[17] Peter B. Ladkin, Using the temporal logic of actions: A tutorial on TLA verification, Technical Report RVS-RR-97-08, RVS group, Universitat Bielefeld, Technische Fakultat, June 1997. Available through http://www.rvs.uni-bielefeld.de

[18] Leslie Lamport, The temporal logic of actions, ACM Transactions on Programming Languages and Systems, vol. 16, No. 3, pp. 872–923, May 1994.

[19] Dirk Henkel, Safely sliding windows, Technical Report RVS-RR-97-05a & 05b, RVS group, Uuniversitat Bielefeld, Technische Fakultat, May 1997.

[20] Dirk Henkel, Safely sliding windows: Into the depths of formal system verification, Research Report, RVS group, Faculty of Technology, University of Bielefeld, April 1999. Available through http://www.rvs.uni-bielefeld.de

[21] Martin Abadi, Leslie Lamport, The existence of refinement mappings, Theoretical Computer Science, 82(2):253-284, May 1991.

[22] Abdelillah Mokkedem, Michael J. Ferguson, and Robert deB. Johnston, A TLA solution to the specification and verification of the RLP1 retransmission protocol, In Proc. Formal Methods Europe Symposium, September 1997.

[23] A. K. Singh and A. K. Bandyopadhyay, A mathematical model for the specification of sliding window protocol, In Proc. CIIC-2001, December 2001, pp. 585–587.