Modeling and Formal Verification of Communication Protocols for Remote Procedure Call

Nilimesh Halder, A.B.M Tariqul Islam, Ju Bin Song

Telecomm. Lab, Dept. of Electronic & Radio Eng.,
Kyung Hee University, Korea

Abstract
This paper presents the modeling and formal verification of some communication protocols for Remote Procedure Call (RPC). These protocols include Request (R) Protocol, Request Reply (RR) Protocol and Request-Reply-Acknowledgement (RRA) Protocol. We have modeled the above-mentioned protocols in Symbolic Model Verifier (SMV), a formal verification tool. In modeling of each protocol, each of the two agents (Client and Server) is modeled as a finite state machine. The common channel between these agents is modeled as a bounded queue of message. Some important features of modeled protocols are then formal verified using the SMV tools.

Key words:
Formal Verification, Symbolic Model Verifier, Remote Procedure Call, Request protocol, Request Reply protocol, Request Reply Acknowledgement protocol.

1. Introduction

Finite state concurrent systems arise naturally in several areas of computer science such as in the design of digital circuits, communication protocols, distributed systems etc. Logical errors found late in the design phase of these systems are an extremely important problem for the circuit designers, protocol designer and the programmers.

Simulation and testing [1] are some of the traditional approaches for verifying the finite state systems. Simulation and testing both involve making experiments before deploying the system in the field. While simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. In the case of circuits and protocols, simulation is performed on the design of the circuit and on that of protocols, whereas testing is performed on the systems themselves. In both cases, these methods typically inject signals at certain points in the system and observe the resulting signals at other points. For software, simulation and testing usually involve providing certain inputs and observing the corresponding outputs. Besides, checking all of the possible interactions and finding potential pitfalls using simulation and testing techniques is not always possible.

Formal verification [2], an appealing alternative to simulation and testing, conducts an exhaustive exploration of all possible behaviors of the system. Thus, when a design is marked correct by a formal verification method, it implies that all behaviors have been explored and the question of adequate coverage or a missed behavior becomes irrelevant. There are some robust tools for formal verification such as SMV, SPIN, COSPAN, VIS, SMART etc [2]. This paper presents the modeling and formal verification of some communication protocols for RPC. We have used SMV as the verification tool.

Manuscript received July 5, 2007
Manuscript revised July 25, 2007

2. Related Works

People have researched formal verification of computer hardware and software for decades. Formal specification of bus protocols (for example, PCI local bus) have been studied widely and extensively [3,4,5,6]. The work in [6,7] promotes a specification style in which the bus protocol is described through an observer, which raises errors signals on the violation of the protocol. A formal verification of the Advanced Micro-controller Bus Architecture (AMBA) protocol from ARM has recently been studied [8]. This protocol has been formally verified using SMV. The Alternating Bit Protocol (ABP) has been verified using formal verification technique [9]. Formal Verification has been also applied to verify the Embedded HW-SW Shared Memory Systems [10]. The bus protocol of the Intel Intenium processor has been formally verified by the SMV model checker [7]. Besides, so many protocols and finite state concurrent systems are being verified using SMV model checker.

3. Formal Verification

Formal verification [2] is used to check if a system holds a property or not. The promise of verification is proving in the sense of mathematical proof, in contrast to conventional simulation and test, which can tell us only that nothing went wrong on the specific case we tried. Obviously, exhaustively trying every possible execution of a system is a valid proof. The formal verification can be viewed as giving the effect of this exhaustive simulation.

3.1 Model Checking

To understand the term *model*, we need to be familiar with transition system and *Kripke Structure*. A transition system is a structure *TS* = (S, S_0, R) where, S is a finite set of states; $S_0 \subseteq$ S is the set of initial states and R $\subseteq$ S$\times$S is a transition relation which must be total i.e. for every s in S there exists s1 in S such that (s, s1) is in R. On the other hand, M= (*S, S_0, R, AP, L*) is a *Kripke Structure*; where (S, S_0, R) is a transition system. AP is a finite set of *atomic propositions* (each proposition corresponds to a variable in the model) and L is a labeling function. It labels each state with a set of atomic propositions that are true in that state. The atomic propositions and *L* together convert a transitions system into a model.

The foremost step to verify a system is to specify the properties that the system should hold. For example, we may want to show that some concurrent program never deadlocks. Once we know which properties are important, the second step is to construct a *formal model* for that system. The model should capture those properties that must be considered for the establishment of correctness.

Each formula representing a property is either true or false in a given state of *Kripke Structure*. Its truth is evaluated from the truth of its sub formula in a recursive fashion, until one reaches atomic propositions that are either true or false in a given states. A formula is satisfied by a system if it is true for all the initial states of the system. Mathematically, say, a *Kripke Structure K= (S, S_0, R, AP, L)* and a formula φ (specification of the property) are given. We have to determine if K | = φ holds (K is a model of φ) or not. K | = φ holds iff K, s | = φ for every s$\in$$S_0$. If the property does not hold, the model checker generally produces a counter example that is an execution path that cannot satisfy that formula.

3.2 Symbolic Model Verifier (SMV)

SMV [11] is a formal verification tool that is used to automatically verify the properties of interacting finite state machines. In SMV, properties are specified in a notation called Computation Tree Logic (CTL), one kind of *temporal logic*. The input language of SMV allows us to describe each of the agents of a system as a module. In particular, the initial states and the transition relation of each of the modules can be specified. SMV constructs a global state transition graph of the entire system from the description of each module. The transition relation and sets of states are viewed as boolean functions. These are represented efficiently by a compact data structure called Binary Decision Diagram (BDD) [12] which involves structure sharing.

3.2.1 Computation Tree Logic (CTL) Formula

Atomic propositions, standard boolean connectives of prepositional logic (e.g., AND, OR, NOT) and temporal operators all together are used to build the CTL formula [13]. Each temporal operator is composed of two parts: a path quantifier (universal (**A**) or existential (**E**)) followed by a temporal modality (***F***, ***G***, ***X***, ***U***) and are interpreted relative to an implicit "current state". There are generally many execution paths (the sequences) of the state transitions of the system starting at the current state. The path quantifier indicates whether the modality defines a property that needs to hold on some paths (denoted by existential path quantifier **E**) or on all paths (denoted by universal path quantifier **A**). The temporal modalities describe the ordering of events in time along an execution path and have the following meaning. (i) ***F*** Ø ("reads 'Ø' holds sometime in the future") is true in a path if there exists a state in that path where formula 'Ø' is true. (ii) ***G*** Ø ("reads 'Ø' holds globally") is true in a path if 'Ø' is true at each and every state in that path. (iii) ***X*** Ø ("reads 'Ø' holds in the next state") is true in a path if 'Ø' is true in the state reached immediately after the current state in the path. (iv) Ø ***U*** φ ("reads 'Ø' holds until 'φ' holds") is true in a path if 'φ' is true in some state in that path, and 'Ø' holds in all preceding states.

3.2.2 Specification of Properties in CTL

CTL formulas are sometime problematical to interpret. For this, a designer may fail to understand what property has been actually verified. Here, we want to add some common constructs of CTL formula. (i) **AG** (*Req* → **AF** *Ack*): it is always the case that if the signal *Req* is true, then eventually *Ack* will also be true. (ii) **AG** (**AF** *DeviceEnabled*): *DeviceEnabled* holds infinitely often on every computation path. (iii) **AG** (**EF** *Restart*): from any state, it is possible to get to the *Restart* state. (iv) **AG** (*Send* → **A** (*Send* **U** *Recv*)): if *Send* holds, then eventually *Recv* is true, and until that time, *Send* remains true. (v) **EF** (*~Ready* ∧ *Started*): It is possible to get to a state where holds *started*, but *ready* does not hold. (vi) **AG** (*in* → **AX AX AX** *out*): Whenever *in* goes high, *out* will go high within three clock cycles. (vii) **AG** (*~storage_coke* → **AX** *storage_coke*): If the coke storage of a vending machine becomes empty, it gets recharged immediately.

3.2.3 Fairness Constraints

In verifying concurrent systems, we are occasionally interested only in correctness along *fair* execution. It is often necessary to introduce some notion of *fairness*. For example, if there are two processes trying to use a shared resource using an arbiter, we may wish to consider only those executions in which the arbiter does not ignore one of its request inputs from either of the processors forever. Alternatively, we may want to consider communication protocols that no message is ever continuously transmitted but never received. A *fairness constraint* can be an arbitrary set of states, usually described by the formula of the logic. If *fairness constraints* are interpreted as a set of states, then a fair path must contain an element of each constraint infinitely often. If *fairness constraints* are interpreted as CTL formulas, then a path is *fair* if each constraint is true infinitely often along the path. The path quantifiers in the logic are then restricted to fair path. A *fairness condition* ***P*** restricts the system to only those paths where ***P*** is asserted infinitely often. Basically the *fairness constraints* are used to rule out undesired executions.

4. Remote Procedure Call (RPC)

RPC is a special case of the general message-passing model of Inter Process Communication (IPC). The primary motivations for developing RPC facility is to provide the programmers with a familiar mechanism for building distributed applications. The RPC mechanism is an extension of the local procedure call in the sense that it enables a call to be made to a procedure that does not reside in the address space of the calling process [14]. The remote procedure may be on the same computer as the calling process or on a different computer.

In case of RPC, since the caller and the callee processes have disjoint address space, the remote procedure has no access to data and variables of the caller's environment. Therefore, the RPC facility uses a message-passing scheme for information exchange between the caller and the callee processes. Fig. 1 shows a typical model of a remote procedure call. The client sends a request message to the server and waits for a reply message. After receiving the request message, the server starts execute the procedure and sends a reply message to the client and wait for next request message. When client receives the reply message, it resumes execution.

Based on the needs of different systems, several RPC communication protocols already exist [15]. These are: (i) The Request Protocol (R Protocol) (ii) The Request /Reply Protocol (RR Protocol) and (iii) The Request /Reply/ Acknowledge-Reply Protocol (RRA Protocol).

4.1 The Request Protocol

This protocol is also known as the R (request) protocol. It is used in which the called procedure has nothing to return as the result of procedure execution and the client requires no confirmation that the procedure has been executed. The client normally precedes execution immediately after the sending the request message, as there is no need to wait for a reply message. The protocol provides call semantics and requires no retransmission of request messages. Fig. 2 shows the message communication of R Protocol. The client sends request message to the server and the server executes the procedure after receiving the request.

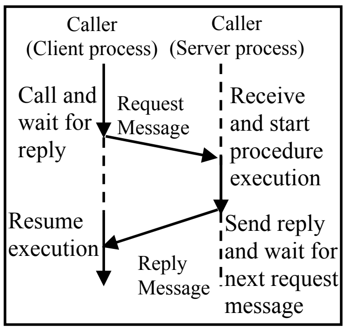


Fig. 1 A typical model of Remote Procedure Call

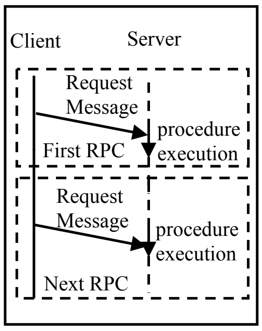


Fig. 2 The Request Protocol

4.2 The Request/Reply Protocol

This protocol is also known as the RR (request-reply) protocol. The protocol is based on the idea of using implicit acknowledgement to eliminate explicit acknowledgement messages. (i) A server's reply message is regulated as an acknowledgement of the client's request message. (ii) A subsequent call packet from a client is regarded as an acknowledgement of the server's reply message of the previous call made by that client. To take care of lost message, timeout based retransmission technique is normally used along with RR protocol. A client retransmits request message if it does not receive the reply message within the predetermined timeout period. Servers can support exactly-once call semantics by keeping records of the replies in a reply cache that enables them to filter out duplicate request message and to retransmit reply messages without the need to reprocess a request. Fig. 3 shows the message communication of RR Protocol. The client sends a request message to the server and waits for a reply message. After receiving the request message, server executes the procedure and also sends a reply message to the client that serves as an acknowledgement for the previous request message. When client receives the acknowledgement from server, it sends next request message that serves as an acknowledgement of previous RPC.

4.3 The Request/Reply/Acknowledge-Reply Protocol

This protocol is also known as RRA (Request/Reply/Acknowledge-Reply) protocol. The RRA protocol requires clients to acknowledge the receipt of reply messages. The server deletes information from its reply cache only after receiving an acknowledgement for it from the client. The RRA protocol provides exactly-once call semantics. In this protocol, there is a probability that the acknowledgement message itself gets lost. Therefore, a unique message identifier is associated with request message. This identifier is also associated with corresponding reply messages. Each acknowledgement message also contains same identifier.

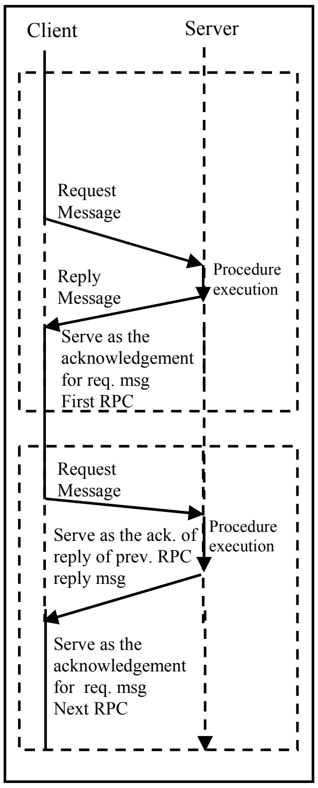

Client
Server
Request Message
Reply Message
Procedure execution
Serve as the acknowledgement for req. msg
First RPC
Request Message
Serve as the ack. of reply of prev. RPC reply msg
Procedure execution
Serve as the acknowledgement for req. msg
Next RPC

Fig. 3 The Request-Reply Protocol

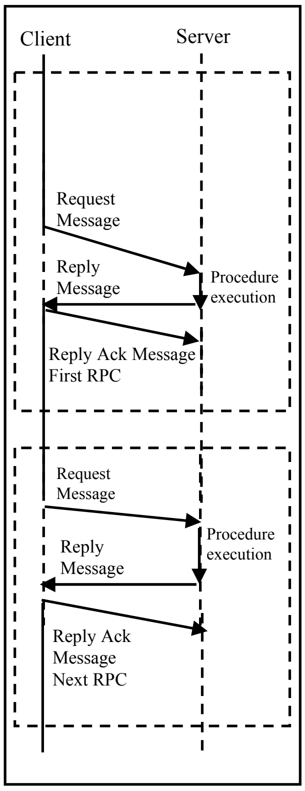


Fig. 4 The RRA Protocol

A client acknowledges reply message to the server only if it has received all previous requests replies. Upon reception of the client's acknowledgement reply for a particular reply result, server deletes reply result entries of all previous requests. So, the loss of any acknowledgement message does not cause any harm to the system. Fig. 4 shows the message communication of RRA Protocol. The client sends a request message to the server and waits for a request message. After receiving the request message, the server executes the procedure and sends a reply message to the client. When client receives the reply message, it sends a reply acknowledge message to the server.

5. Modeling of The Communication Protocols

The Cadence SMV has been used to formally verify the communication protocols for RPC. The aim of this modeling and verification is to check if the protocols hold all the desired properties and do not hold any undesired property. In order to verify any protocol formally, it is very much important to model that protocol carefully. In this section, we have discussed our modeling of protocols in SMV.

5.1 Power Game Model of Network

Firstly, each of the two agents (client and server) of each of the communication protocols (R, RR and RRA) is modeled as finite state machine. To establish communication between client and server for each protocol, a reliable communication channel is required. Here, this communication channel is modeled as a queue of message, which is the integral part of both client and server of each protocol.

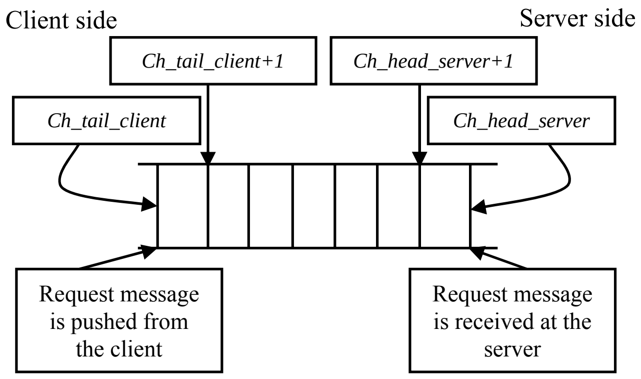

Client side
Server side
Ch_tail_client+1
Ch_head_server+1
Ch_tail_client
Ch_head_server
Request message is pushed from the client
Request message is received at the server

Fig. 5 Modeling of communication channel as a queue of request message.

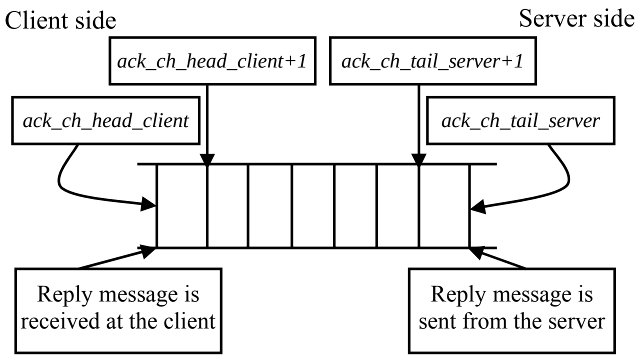


Fig. 6 Modeling of communication channel as a queue of reply message.

Fig. 5 shows the modeling of communication channel as a queue of request message. The request message is pushed through the tail of the queue from the client side and the request message is received from the head of the queue at the server side. Fig. 6 shows the modeling of communication channel as a queue of reply message. The reply message is sent from the server side and it is received from the head of the queue at client side. The brief description of modeling of client and server of each of the protocols in SMV are given in the following sections.

5.1.1 Modeling of R Protocol

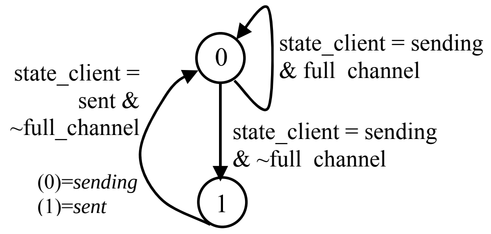
In R protocol, the communication channel is used only for sending request message from client to server. There is nothing about acknowledgement. The communication channel is modeled as a queue of fixed length. This queue is modeled as message queue. The client module and the server module of R protocol are described as follows.

5.1.1.1 Client Module

In the *client*, the tail and head of the queue are denoted as *ch_tail_client* and *ch_head_server* respectively. The variable full_channel in client module is used to check the status of the queue by statement (*full_channel* := ((*ch_tail_client* + 1) mod *Q_SIZE*) = *ch_head_server*;). If the queue is full then the client has to wait until the queue is not free to send request message. Otherwise, the client sends request message increasing the tail. The state variable *state_client* can hold either of the value: {*sending* or *sent*}. The variable *sending* means that the client is in the state of sending message to server via queue. The variable sent means that the client is in the state that it already sends message to server through the queue. Initially, the client is in sending state. The client checks the statement *state_client = sending & ~full_channel* to send message. If it is true then the tail is increased by the statement next (*ch_tail_client*):= (*ch_tail_client*+1) mod *Q_SIZE*; and the client moves to state *sent*. After a fixed time period the client returns in state *sending*. Fig. 7 shows the states of client for request message. If the state of client is sent and the channel is not full, it changes its state from 1 to 0. On the other hand, if the state is sending and the channel is full, it remains at state 0, but if the channel is not full, it changes its state from 0 to 1.

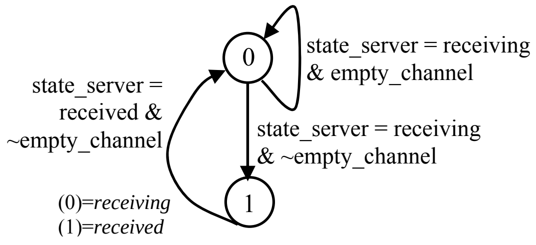
5.1.1.2 Server Module

The server module maintains two variable *ch_head_server* and *ch_tail_client* for the queue. The variable *empty_channel* is used to check the status of the queue for the server by the statement (*empty_channel* := (*ch_tail_client* = *ch_head_server*);). If the queue is empty then the server does not change its state. The state variable of server is *state_server* that can hold either of the value: {*received* or *receiving*}. Initially, the server is in receiving state. The server checks the condition (*state_server = receiving & ~empty_channel*) to receive message from client. If it is true then the server executes the statements (next (*ch_head_server*) := (*ch_head_server*+1) mod *Q_SIZE*; and next (*state_server*):=*received*;). These mean that the message is received from the queue and the server changes its state to received. Fig. 8 shows the states of server for request message. If the state of server is received and the channel is not empty, it changes its state from 1 to 0. On the other hand, if the state is receiving and the channel is empty, it remains at state 0, but if the channel is not empty, it changes its state from 0 to 1.



0
1
state_client = sending & full channel
state_client = sending & ~full channel
state_client = sent & ~full_channel
(0)=sending
(1)=sent

Fig. 7 States of client for request message



0
1
state_server = receiving & empty_channel
state_server = receiving & ~empty_channel
state_server = received & ~empty_channel
(0)=receiving
(1)=received

Fig. 8 States of server for request message

5.1.2 Modeling of RR Protocol

In RR protocol the communication channel is used for sending request message from client to server and for receiving reply message (acknowledgement) from server to client. The communication channel is modeled as a queue of fixed length for both request and reply message from client and server respectively. The client module and the server module of RR protocol are described as follows.

5.1.2.1 Client Module

The modeling client module of RR protocol is similar to that of R protocol in case of sending message from client to server. Here the message queue can hold only one message at a time. This ensures that one message can be sent at a time and the next message can be sent after having the acknowledgement for the previous one from the server. The client module of RR protocol uses two variables *ack_ch_head_client* and *ack_ch_tail_server* for the modeling of the queue of reply message. The client module uses the variable *empty_ack_channel* to check whether the queue either full or not by the statement (*empty_ack_channel* := (*ack_ch_head_client* = *ack_ch_tail_server*);). The variable *state_client_for_ack* can either hold *received* or *receiving*. The client checks the condition (*state_client_for_ack* = *receiving* & ~*empty_ack_channel*) to receive reply message. If it is true then the client pops the reply message by increasing the *ack_ch_head_client* and changes its state to *received*. Fig. 9 shows the states of client for request message and Fig. 10 shows the states of client for reply message. It is shown in Fig. 9 that if the state of client for send is *sent* and the channel is not full, it changes its state from 1 to 0. On the other hand, if the state is *sending* and the channel is full, it remains at state 0, but if the channel is not full, it changes its state from 0 to 1. It is also shown in Fig. 10 that if the state of client for acknowledgement is *received* and the acknowledgement channel is not empty, it changes its state from 1 to 0. On the other hand, if the state is *receiving* and the acknowledgement channel is empty, it remains at state 0, but if the channel is not empty, it changes its state from 0 to 1.

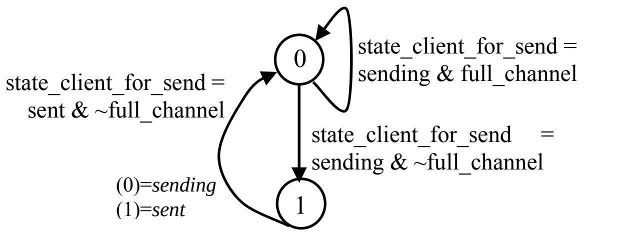

state_client_for_send = sending & full_channel
state_client_for_send = sent & ~full_channel
state_client_for_send = sending & ~full_channel
0
1
(0)=*sending*
(1)=*sent*

Fig. 9 States of client for request message

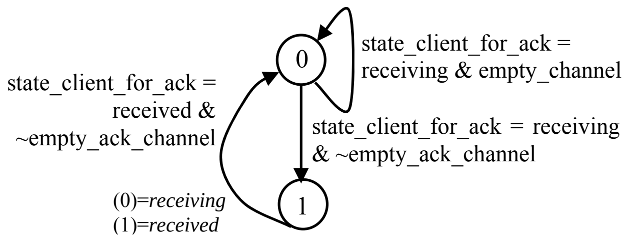

state_client_for_ack = receiving & empty_channel
state_client_for_ack = received & ~empty_ack_channel
state_client_for_ack = receiving & ~empty_ack_channel
0
1
(0)=*receiving*
(1)=*received*

Fig. 10 States of client for reply message

5.1.2.2 Server Module

The modeling server module of RR protocol is similar to that of R protocol for receiving message. The server module uses the variable *ack_ch_tail_server* and *ack_ch_head_client* to indicate the tail and head of the queue of reply message. The *full_ack_channel* is used to check whether the queue is full or not by the statement (*full_ack_channel*:=((*ack_ch_tail_server*+1)mod *Q_SIZE*) =*ack_ch_head_client*;).The *state_server_for_ack_send* of server can hold either *sent* or *sending* state. The server checks the condition *state_server_for_ack_send* = *sending* & ~*full_ack_channel*. If the condition is true, the server increases the value of *ack_ch_tail_server* to indicate that the reply message is sent and changes its state to *sent*. Otherwise, it is in its same state. Besides this, the server is in sending state for any other condition. Fig. 11 shows the states of server for request message and Fig. 12 shows the states of server for reply message. It is shown in Fig. 11 that if the states of server for receive is *received* and the channel is not empty, it changes its state from 1 to 0. On the other hand, if the state is *receiving* and the channel is empty, it remains at state 0, but if the channel is not empty, it changes its state from 0 to 1. It is also shown in Fig. 12 that if the state of server for send is *sent* and the acknowledgement channel is not full, it changes its state from 1 to 0. On the other hand, if the state is *sending* and the acknowledgement channel is full, it remains at state 0, but if the channel is not full, it changes its state from 0 to 1.

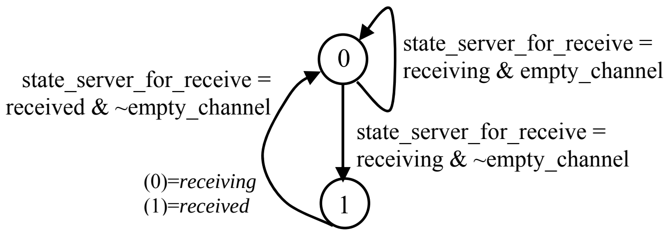

state_server_for_receive = receiving & empty_channel
state_server_for_receive = received & ~empty_channel
state_server_for_receive = receiving & ~empty_channel
0
1
(0)=*receiving*
(1)=*received*

Fig. 11 States of server for request message

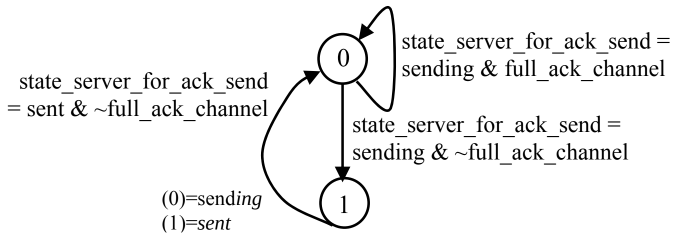

state_server_for_ack_send = sending & full_ack_channel
state_server_for_ack_send = sent & ~full_ack_channel
state_server_for_ack_send = sending & ~full_ack_channel
0
1
(0)=send*ing*
(1)=*sent*

Fig. 12 States of server for reply message

5.1.3 Modeling of RRA Protocol

In RRA protocol the communication channel is used for sending request message from client to server, for receiving reply message (acknowledgement) from server to client and for sending reply acknowledgement message from client to server. The communication channel is modeled as a queue of fixed length for request message, reply message and reply acknowledgement message. The client module and the server module of RRA protocol are described as follows.

5.1.3.1 Client Module

The modeling of request message and reply message of the RRA protocol is similar to that of the RR protocol. The client uses *reply_ack_ch_tail_client* and *reply_ack_ch_head_server* to indicate the tail and head of the queue of reply acknowledgement message. The modeling of this queue is like to the modeling of the message queue of R protocol. The variable *full_reply_ack_channel* is used to check that the queue is either full or not by statement (*full_reply_ack_channel*:=((*reply_ack_ch_tail_client*+1) mod *Q_SIZE*) = *reply_ack_ch_head_server*;). The variable *state_client_for_reply_ack* holds either *sent* or *sending*. This module also uses two variables *ack_test* and *reply_ack_test*. The *ack_test* and *reply_ack_test* are used as flag variables for reply message and for reply acknowledgement message respectively. Fig. 13 shows the states of client for reply acknowledgement message. If the state of client for reply acknowledgement is *sent* and the acknowledgement channel is not full, it changes its states from 1 to 0. If the state of client for reply acknowledgement is *receiving* and the acknowledgement channel is empty, it remains 0 state, but if acknowledgement channel is not empty, it changes its state from 0 to 1.

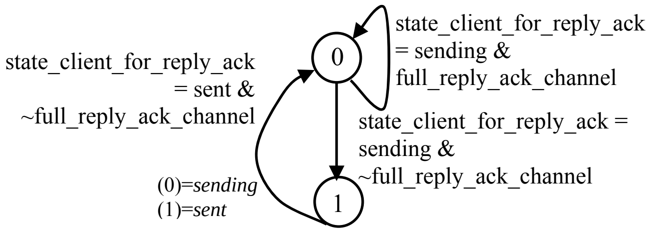

state_client_for_reply_ack = sent & ~full_reply_ack_channel
0
state_client_for_reply_ack = sending & full_reply_ack_channel
state_client_for_reply_ack = sending & ~full_reply_ack_channel
(0)=sending
(1)=sent
1

Fig. 13 States of client for reply acknowledgement message

5.1.3.2 Server Module

The modeling of receiving request message and sending reply message of the RRA protocol is similar to that of the RR protocol. The modeling of reply acknowledgement message of the server is similar to that of request message of the server. To do this, the server module uses the variables *reply_ack_ch_head_server* and *reply_ack_ch_tail_client*. The *empty_reply_ack_channel* checks whether the channel is either empty or not. The state variable is *state_server_for_reply_ack*. The server module uses the variable *memory_cache* that can take either full or free and a boolean variable *cache_test* for modeling of reply result of reply cache of server. After sending the reply message the server module changes the initial state of *memory_cache* to full for each request message and the boolean *cache_test* is set. When the server receives reply acknowledgement message from the client the *memory_cache* is free and *cache_test* is reset. Thus after having the reply acknowledgement the server deletes its reply cache. Fig. 14 shows the states of server for reply acknowledgement message. If the state of server for reply acknowledgement is *received* and the reply acknowledgement channel is not empty, it changes its states from 1 to 0. If the state of server for reply acknowledgement is *receiving* and the reply acknowledgement channel is empty, it remains 0 state, but if reply acknowledgement channel is not empty, it changes its state from 0 to 1.

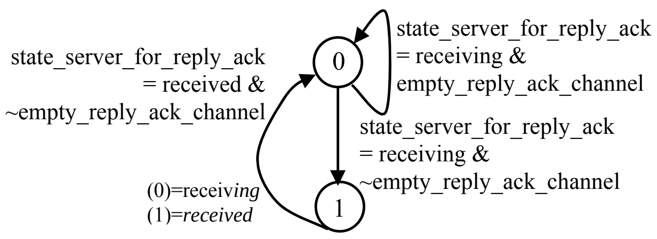

state_server_for_reply_ack = received & ~empty_reply_ack_channel
0
state_server_for_reply_ack = receiving & empty_reply_ack_channel
state_server_for_reply_ack = receiving & ~empty_reply_ack_channel
(0)=receiving
(1)=received
1

Fig. 14 States of server for reply acknowledgement message

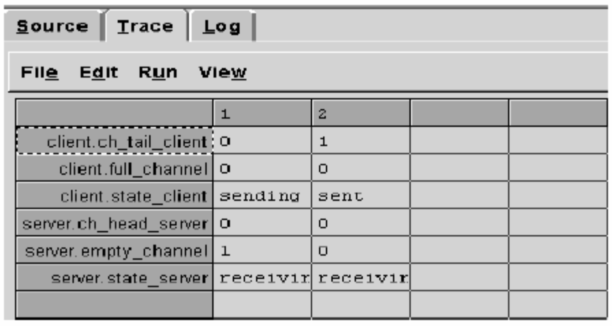
6. Formal Verification / Result

The results of the verification of the protocols are stated below.

6.1 Properties for R Protocol

a) SPEC **AG** (*client.state_client* = *sent* → **AF**(*server.state_server*=*received*)); means that once the client sends the request message, eventually the server receives it. The SMV shows that this property holds.
b) SPEC **AG** (*client.state_client*=*sent* → **AF** (*server.state_server*=*received*)); means that if the request message has been already sent, then eventually the server receives it. The SMV shows that this property holds.
c) SPEC **AG** *server.empty_channel*; means that the massage queue is always empty. The SMV shows that this property does not hold.

Since the property does not hold, the SMV produces a counter example, which shows the main reason of not holding the property specified. The generated counter example by SMV is shown in Fig. 15.



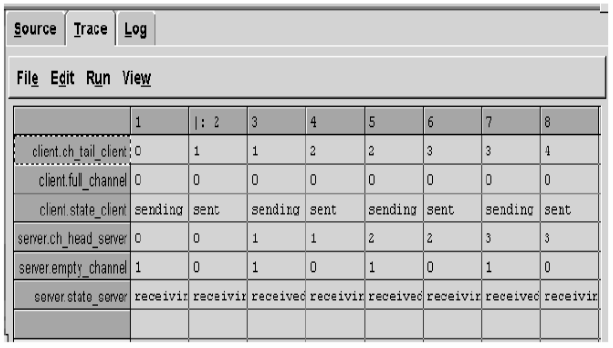
	1	2		
client.ch_tail_client	0	1		
client.full_channel	0	0		
client.state_client	sending	sent		
server.ch_head_server	0	0		
server.empty_channel	1	0		
server.state_server	receivir	receivir		

Fig. 15: Counter example for SPEC AG *server.empty_channel*

The counter example shows two states of not holding the specified property. The client and server module of R protocol has several variables for modeling of the protocol. These variables take different signals. Here it is clear that when client will be in sending state, client.full_channel will not be full and server will be in receiving state. Then the client will send request message and the server channel will not be empty. Thus the specification is not true. This is shown in column 1 of Fig. 15. In the column 2 of Fig. 15, when state of client will have been sent then the server channel will not be empty.

d) SPEC **AF** *client.full_channel*; means that the message queue will be full in every execution path at some time in future. The SMV shows that this property does not hold.

Since the property does not hold, the SMV produces a counter example, which shows the main reason of not holding the property specified. SMV generates the following counter example as shown in Fig. 16.



	1	|: 2	3	4	5	6	7	8
client.ch_tail_client	0	1	1	2	2	3	3	4
client.full_channel	0	0	0	0	0	0	0	0
client.state_client	sending	sent	sending	sent	sending	sent	sending	sent
server.ch_head_server	0	0	1	1	2	2	3	3
server.empty_channel	1	0	1	0	1	0	1	0
server.state_server	receivir	receivir	receivec	receivir	receivec	receivir	receivec	receivir

Fig. 16: Counter example for SPEC AF *client.full_channel*

Here, it is clear that when *server.empty_channel* is true i.e. server channel is empty, state of client will be *sending* and state of server will be *receiving*. Then the client must send request message through the channel. Thus the client message queue is not always full in future. The result is shown in column 1 of Fig. 16. The remaining columns of the above figure shows that in future the message queue will not be always full because the client will send a request message by increasing *client.ch_tail_client* and this message will be received by the server. So, message queue will not be always full at sometime in future.

6.2 Properties for RR Protocol

a) SPEC **AG** ((*client.state_client_for_send*=*sent* & *msg*=*message*)→**AF**(*client.state_client_for_ack*=*received*)); means that once the client sends the request message (*msg*), eventually it receives acknowledgement from the server. The SMV shows that this property holds.

b) SPEC **AG** (*client.state_client_for_ack*=*received*& *msg*=*ack_message*)→**AF**(*server.state_server_for_receive* = *receiving* & *client.state_client_for_send*=*sent* & *msg*=*message*); means that if the client receives acknowledgement (*msg*) and the client sends the next message, the server is eventually in receiving state for that message. The SMV shows that this property holds.

6.3 Properties for RRA Protocol

a) SPEC **AG** ((*client.state_client_for_send*=*sent* & *msg*=*message*)→**AF**(*client.state_client_for_ack*=*received*)); means that once the client sends the request message (*msg*), eventually it receives acknowledgement from the server. The SMV shows that this property holds.

b) SPEC **AG** ((*client.state_client_for_send*=*sent* & *msg*=*ack_message*&*client.state_client_for_ack*=*received*) → **AF**(*server.state_server_for_reply_ack*=*received*)); means that once if the client sends the request message and gets the reply message for that then the server receives reply acknowledgement from the client eventually. The SMV shows that this property holds.

c) SPEC **AG** (*server.state_server_for_reply_ack* = *received*) → **AF** (*server.memory_cache*=*free*); means that once the server receives reply acknowledgement from the client then it deletes all previous results from its reply cache eventually. The SMV shows that this property holds.

7. Conclusion

Verification techniques are useful in automatically detecting subtle corner cases of the protocol specifications. In this paper, our experience in verification of the communication protocols for RPC using SMV is presented. Here, we have verified some most common properties of the communication protocols and found that the properties hold. Thus from our experience we can say that the protocols are reliable for communication in distributed system. Our next aim is to verify these protocols using other verification tool such as SPIN. Moreover, some timing constraints can also be imposed to verify the protocols.

References

[1] G. J. Myers. The Art of Software Testing. Wiley, 1979.
[2] Edmund M. Clakre, Jr. Orna GrumBerg and Doron A. Peled, " Book: Model Checking", The MIT Press; Second Printing 2000.
[3] F. Aloul and K. Sakallah, "Efficient verification of the PCI local bus using Boolean satisfiability". International Workshop on Logic Synthesis (IWLS), 2000.
[4] P. Chauhan, E. Clarke, Y. Lu and D. Wang, "Verifying IP-core based System-on-Chip design", IEEE ASIC SOC conference, 1999.
[5] A. Mokkedem, R. Hosabettu, M. Jones and G. Gopalkrishnan, "Formalization and analysis of a solution to the PCI 2.1 bus transaction ordering problem", Formal Methods in System Design, 16, 2000.
[6] K. Shimuzu, D. Dill and A hu, "Monitor based formal specificationof PCI", International Conference on Formal Methods in computer Aided Design (FMCAD), 2000.
[7] K. Shimuzu, D. Dill and C. T. Chou, "A specification methodology by a collection of compact properties as pipelined to the Intel Itanium processor bus protocol ". Correct Hardware design and Verification Methods (CHARME),LNCS 2144, 2001.
[8] Abhik Roychoudhury, Tulika Mitra and S. R. Karri , " Using formal techniques to the debug the AMBA system on chip Bus Protocol", IEEE/ACM conference on design automation and Test in Europe (DATE), 2003.
[9] Kamrul Hasan Talukder, " Formal verification of the Alternating Bit Protocol", 6th International Conference on computer & Information Technology (ICCIT) 2003, Dhaka,Bangladesh.
[10] Om Prakash Gangwal, Andre Nieuwland, Paul Lippens, "A Scalable and Flexible Data Synchronzation Scheme For Embedded HW-SW Shared Memory Systems." Embedded Systems Architectures on Silicon, Philips Research Laboratories, The Netherlands.
[11] Cadence Berkeley Laboratories, Free download from http:// www-cad.eecs.Berkeley.edu/ ~kenmcmil/smv/, Califonia, USA. The SMV Model Checker, 1999.
[12] R.Bryant, "Graph-based algorithms for boolean function manipulation", IEEE Transactions on Computers, C-35, No.8, 1986, pp.677-691.
[13] Kamrul Hasan Talukder and MD. Khademul Islam Molla, "Computation Tree Logic in model checking",3rd International IT conference, Nepal, 2003.
[14] George Coulouris "Distributed Systems Concepts and Design", Pearson Education Asia.
[15] Birrell, A.D., and Nelson, B., "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol. 2, No.1, pp. 35-39 (1984).

Nilimesh Halder is currently a PhD student and a member of Telecom. Lab in the Dept. of Electronic and Radio Eng. at Kyung Hee University. He received the B.S. degree in Computer Science and Engineering from Khulna University, Bangladesh in 2005. His research interests include wireless communications, ad-hoc mobile networks, radio resource management, power control & management, self-organizing networks, cognitive radio and game theory.

A.B.M Tariqul Islam is currently a PhD student and a member of Telecom. Lab in the Dept. of Electronic and Radio Eng. at Kyung Hee University. He received the B.S. degree in Computer Science and Engineering from Khulna University, Bangladesh in 2005. His research interests include ad-hoc mobile networks, radio resource management, wireless communications, wireless sensor networks, cognitive radio and spectrum sensing.

Ju Bin Song is currently a professor in the Dept. of Electronic and Radio Eng., Kyung Hee University, from 2003. He received BS and MS degree in 1987 and 1989, respectively and PhD degree in the Department of Electronic and Electrical Eng., University College of London, UK in 2001. He was senior researcher in ETRI from 1992 to 1997 and a research fellow in UCL, 2001. He was a professor in Hanbat National University from 2002 to 2003. His research interests include telecommunications, mobile multi-hop networks, radio resource management, next generation communications. He is a member of IEEE, KICS and KIEE.