Research Journal of Applied Sciences 3 (4): 288-293, 2008
ISSN: 1815-932X
© Medwell Journals, 2008

Design and Implementation of a Model of a Specification Language for Formal Verification

Kamrul Hasan Talukder, Ahmed Shah Mashiyat and Rezoanoor Rahman
Computer Science and Engineering Discipline, University of Khulna, Khulna, 9208, Bangladesh

Abstract: The world is moving towards automation and automation requires correctness. So the importance of formal verification tools is increasing rapidly as it is an automatic technique for verifying finite state concurrent systems. However, the industrial usability of formal verification tools remains limited for the complexities and expertise needed for modeling the behavior of a system. In this study, we present a methodology that can model the behavior of a system in Textual Specification Language (TSL) based on Message Sequence Charts (MSC's) with less expertise and effort. For proper illustration of TSL, we convert the TSL specification into Symbolic Model Verifier (SMV) code using a Turbo C/C++ program and then verify some properties of the system expressed in Computational Tree Logic (CTL) with the help of the SMV model checker, producing verdicts or counter example.

Key words: Formal verification, model checking, textual specification language, symbolic model verifier, message sequence charts, computational tree logic

INTRODUCTION

In recent years, hardware and software systems are widely used in every day life. The involvement of these systems with human life is increasing rapidly. So, the correctness of these systems is very important. Logical errors found late in the design phase of these systems are an extremely crucial problem for both designers and programmers. These kinds of errors are very hard to detect through informal reasoning like simulation and testing, especially when the number of possible states of the system is very large. Formal verification, an appealing alternative to simulation and testing, explores all the possible behaviors of the system to be verified while the simulation and testing explores some but not all the possible scenarios of the system. Thus there is no doubt about the result produced by formal verification method. For this reason, the application of formal verification is increasing day-by-day and getting popular in industry (Holt, 1999; Talukder, 2003a, b; Halder *et al*., 2005). Some robust tools, such as SPIN, SMV, COSPAN, VIS and SMART are widely used for formal verification.

In spite, of the huge amount of work done by the researchers in this field, there still exits some gap between the industry and the research community for proper automation of formal verification tools. Our view is that one key abstraction for this situation is the expertise and complexities needed to specify a system with the exiting modeling language.

So, our main objective is to reduce the effort and complexity to specify a system behavior by the conventional specification language like SMV, SPIN, COSPAN and VIS. For this reason we develop a specification language in which the system behavior is easy to code and the effort required to specify the system is much less. For comparison we chose the well-known formal verification language SMV. A system specified by the Textual Specification Language (TSL) is converted to SMV code and then verifies by the SMV model Checker.

OVERVIEW OF PROPOSED SYSTEM

The main objective of this research is to develop a specification language for Formal Verification of communicating components. We consider a simple bus protocol as the communicating components as an example. At the very first step, our system models the total system using the MSC's, a popular tool for formally explaining the behavior of the functional component within the environment. From the MSC's the model is re-specified with Textual Specification Language (TSL) following a newly developed grammar. This part of the system should be done informally. Then the TSL file is converted to SMV code. The converter includes a parser developed by well-known parser generator java cup 0.10 along with lexical analyzer tool jflex1.4.1. We have developed the translator with C\C++. The properties of the bus protocol to be verified are expressed in CTL

Corresponding Author: Ahmed Shah Mashiyat, The Jaxara IT Ltd., Dhaka-1213, Bangladesh

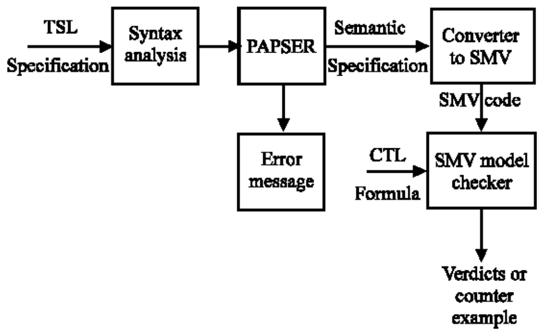


Fig. 1: System architecture

formulas and embedded with the generated SMV code. The SMV Model Checker producing verdicts or counter example then verifies this code. The system architecture is given in Fig. 1.

TSL: A NEW SPECIFICATION LANGUAGE

In course of the development of the system, we have formalized a new specification language called Textual Specification Language (TSL) for modeling the whole behavior of the communication components. The TSL specifies a model consists of a set of process where each process will represent a system component or an interface. A transaction scheme is the unit of interaction between the processes. The transaction scheme is consists of a collection of transactions where a transaction is modeled as an MSC. An execution of a transaction scheme will choose one of its transactions depending on the current values of the guard of the participating processes. A system specification in TSL consists of a sequence of declarations like *process, scheme, transaction, agents, guard, send and recv etc.* The TSL specification example of the simple bus protocol in the stucy 4 will give the reader a clear illustration of the structure of TSL language model. The grammar for the new language is stated below:

```
S                  →  VERIFY pro-name { proc-dec-part
                      trans-sch-dec-part }
proc-dec-part      →  proc-dec-part  proc-dec | proc-dec
proc-dec           →  PROCESS process-name {
                    var-dec-part
                    equation-part  }
type-id            →  basic-type | enum-type | array-type
basic-type         →  range-type | BOOLEAN
enum-type          →  {id-list }
id-list            →  identifier (, identifier)+
array-type         →  ARRAY range-type OF basic-type
range-type         →  interger-const .. interger-const
var-dec-part       →  var-dec-part var-dec; | →
var-dec            →  type-id : id-list
equation-part      →  EQUATION id-list;
trans-sch-dec-part →  trans-sch-dec trans-sch-dec-part |
                      trans-sch-dec-part }
 trans-sch-dec     →  SCHEME trans-schm-name { trans-
                      list }
trans-list         →  trans-dec trans-list | trans-dec
trans-dec          →  TRANSACTION trans-name {
                      AGENTS { agent-list } guard-
                      section  }
                      | TRANSACTION trans-name {
                      AGENTS { agent-list } }
guard-section      →  GUARD guard ;
agent-list         →  agent-list agent | agent
agent              →  process-name : event-list
event-list         →  event , event-list | event ;
event              →  send(mesg-id, var) | send(mesg-id,
                      const, type) |
                      recv(process-name.mesg-id) |
                    {action}
action-atom        →  simple-stmt ; | if-stmt
action             →  action action-atom | action-atom
simple-stmt        →  var := expr | var :=DIN
expr               →  expr [* | / | and] F | F
F                  →  F + G | F - G | F | G
G                  →  G mod H | H
H                  →  H RelOp I | I
I                  →  ~I | -I | (expr) | var | const
const              →  integer-const | boolean-const
integer-const      →  integer-const digit | digit
digit              →  [0..9]
boolean-const      →  0 | 1
guard              →  guard and guard-atom | guard-atom |
                      (guard-atom)
guard-atom         →  ~prop | prop
prop               →  prop or prop-atom | prop-atom
prop-atom          →  scoped-var relop const | scoped-var
                      relop scoped-var |
                      scoped-var
relop              →  = | < | > | ≤ | ≥ | !=
if-stmt            →  IF expr { action } | IF expr action-
                      atom | IF expr {action}
                      ELSE {action}
var                →  identifier | identifier[identifier] |
                      identifier [integer-const]
scoped-var         →  process-name.var
pro-name           →  identifier
process-name       →  identifier
trans-name         →  identifier
mesg-id            →  identifier
trans-schm-name    →  identifier
```

A detail description of this grammar syntax is out of the scope of the current study.

A CASE STUDY

Modeling A system by MSC's: For modeling a system behavior with TSL, we first model the system by MSC's. MSC is a trace language which in its graphical form admits a particularly intuitive representation of system runs in distributed systems while focusing on the message interchange between communicating entities and their environment (Rudolpha *et al*., 2008). Due to their intuitive notation MSC's have been proven useful as a communication or collaboration tool among a set of

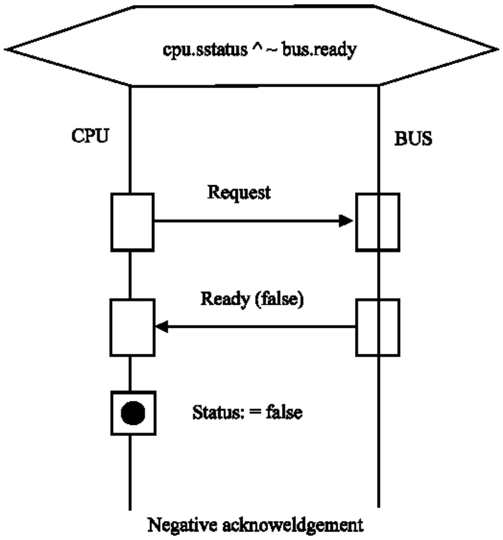


Fig. 2a: Negative acknowledgement by bus (slave)

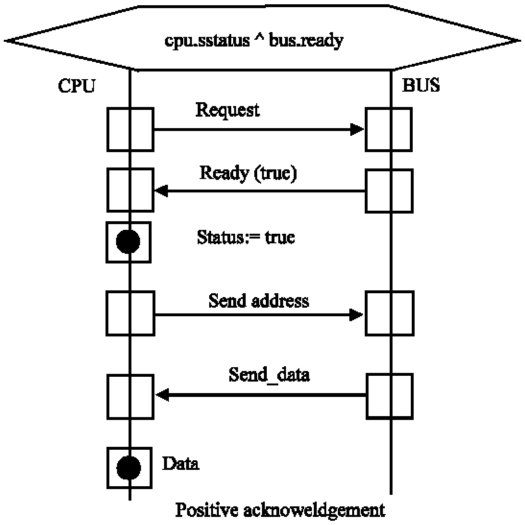


Fig. 2b: Positive acknowledgement by bus (slave)

components, thus helping to reduce misunderstandings from the very early development stages.

This step is not a must but as the TSL is a new model language, this step will help to realize the intuitive notion of TSL. For modeling a system let us consider the PCI local bus as an example (Mokkedem *et al.*, 2000; Aloul and Sakallah, 2000). PCI is a high performance synchronous bus standard developed by Intel. A PCI bus transaction involves a master (initiator) and a slave (target) device. When a device wants to perform a particular transaction, it acquires the bus and specifies what type of transaction will be carried out next.

Here we consider only a single transaction that took place between a CPU (master) to BUS (*salve*) to send and receive data. The transactions can be either of negatively or positively acknowledged based on the status of the BUS. When CPU wants to receive data from bus, it first sends request to BUS. In return BUS sends either negative or positive acknowledgement to the CPU, based on the status variable, *bus.ready*.

If the acknowledgement is negative, the CPU stops attempting to receive data. And the transaction stops. The MSC of the negatively acknowledged transaction is shown in the Fig. 2 a.

Again, if the BUS is ready to send data, it sends positive acknowledgement, when *bus.ready* is true. The CPU then sends address to the BUS. Sequentially, the BUS sends the desired data to be transmitted, which is stored at CPU. The corresponding MSC can be shown as the Fig. 2 b.

Modeling a system with TSL: The MSC's discussed in the previous study corresponds to two functioning component. Two transactions, named as positively and negatively acknowledged, are took part in the transaction scheme. The timing sequences of the MSC's Show that there are three actions under CPU process and two actions under BUS process in the negatively acknowledged transaction. Again there are five actions under CPU process and four actions under BUS process in positively acknowledged transaction. The CPU-BUS interaction is modeled in SL form in the following example according to the TSL grammar.

Parsing and code generation: As we have said earlier that we will translate the TSL specification code into SMV code for proper illustration of the inherent potential of the TSL. The translation procedure from TSL to SMV code is completely automated. The translator consists of a lexical analyzer, a parser and a converter. Jflex-1.4.1 is used for the development of the lexical analyzer.

```
BusCpu Example {
process cpu {
        0..7 : addr_buf;
        0..16 : data_buf;
        boolean : status;
        equation T1;
}
process bus {
        boolean : ready;
        0..7 : addr_buf;
        0..15 : data_buf;
        equation T1;
}
scheme T1 {
      transaction nack {
         agents {
        cpu : send(req,1,boolean),
               recv(bus.ack),
               {status:=din;};
        bus : recv(cpu.req),
              send(ack,0,boolean);
```

```
          }
        guard ~bus.ready  and cpu.status;
      }
  transaction pack {
        agents {
        cpu : send(req,1,boolean),
                    recv(bus.ack),
              {status:=din;},
              send(addr,1,boolean),
               recv(bus.data);
        bus : recv(cpu.req),
                    send(ack,1,boolean),
              recv(cpu.addr),
              send(data,data_buf);
            }
        guard bus.ready and cpu.status;
    }
   }
}
```

The parser is developed with the Java CUP parser generator version 0.10. CUP is a system for generating Look-Ahead LR (LALR) parsers from simple specifications. The parser produces the semantic representation of the specification. The semantic representation of the specification is then converted to the SMV code.

As the whole mapping technique of the system is beyond the scope of this presentation, we have discussed only a general overview of the code generation procedure based on the example stated in the study. Each process of a TSL specification corresponds to a module of the SMV code except the "main" module. For each pair of communicating process, there are 2 channels. Hence the number of the channel queue should be n*(n-1) for n number of processes so that the processes are fully connected. Each module should contain 2*n*(n-1) pointer variables to indicate heads and tails of the queue channels.

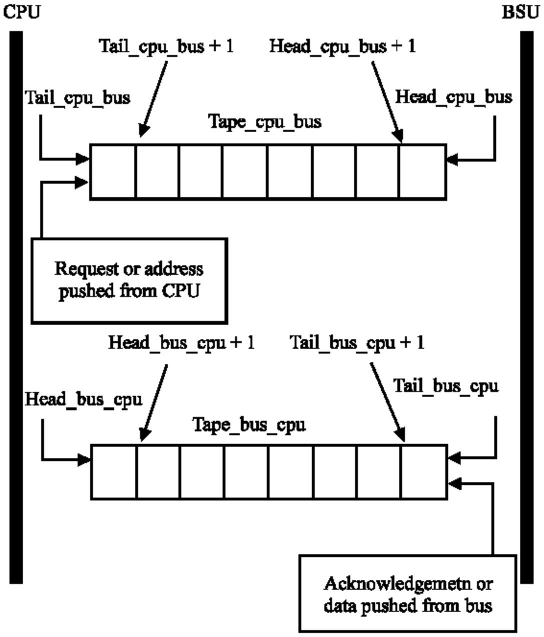
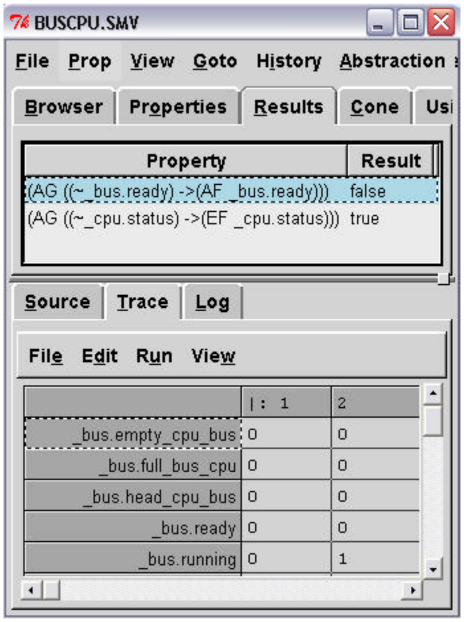


Fig. 3: Communication channel between two processes

Eventually there are some Boolean variables that will indicate whether the queues are full or empty and some actions to measure them. The original variables of each process are also included within the module. The number of states of the SMV code corresponds to the number of identical sequential actions of a particular process within the whole transaction scheme. The action execution and state transition of each state is guarded with some local variables. Figure 3 will help the code generation procedure of the example. Now let us consider the 'BusCpu' example stated in the study 2. There are two processes (CPU and BUS) in the example along with some local variables and some transactions between them. So, 2 modules will be produced. Here we have described about only the CPU module. As the local variables of each process in TSL will be in the corresponding module, CPU will have the variables for status, add_buff and data_buff. The CPU module uses two pointers head_cpu_bus and tail_cpu_bus for request or address sending queue and two pointers head_bus_cpu and tail_bus_cpu for acknowledgement and data receiving queue. This module also uses the variable empty_cpu_bus and empty_bus_cpu to check whether the queues are empty or not by the statements empty_bus_cpu := (head_bus_cpu = tail_bus_cpu) and empty_cpu_bus := (head_cpu_bus = tail_cpu_bus). The variables full_bus_cpu and full_cpu_bus check whether the queues are full or not by the statements full_bus_cpu := ((tail_bus_cpu+1) mod Q_SIZE) = head_bus_cpu and full_cpu_bus:= ((tail_cpu_bus+1) mod Q_SIZE) = head_cpu_bus. The first three actions of the CPU process under different transactions are identical. So the number of states for the 6 actions will be three. Since the other two are not the same they will produce two more states. So total number of states in the module will be five. We numbered them as _cpu0, _cpu1, _cpu2, _cpu3, _cpu4. The initial state is _cpu0. The action "send<req, 1>" will be executed only if full_cpu_bus is not true and the current state is _cpu0. The action "recv<bus.ack>" will be executed only if empty_bus_cpu is not true and the current state is _cpu1. If the current state is _cpu2 then the action "status:=din.bool" will be executed. The action "send<add, 1, boolean>" will be executed only if full_cpu_bus is not true and the current state is _cpu3. The action "recv<bus.data>" will be executed only if empty_bus_cpu is not true and the current state is _cpu4


BUSCPU.SMV
File Prop View Goto History Abstraction
Browser Properties Results Cone Usi
Source Trace Log
File Edit Run View

Property	Result
(AG ((~_bus.ready) ->(AF _bus.ready)))	false
(AG ((~_cpu.status) ->(EF _cpu.status)))	true

	| : 1	2
_bus.empty_cpu_bus	0	0
_bus.full_bus_cpu	0	0
_bus.head_cpu_bus	0	0
_bus.ready	0	0
_bus.running	0	1

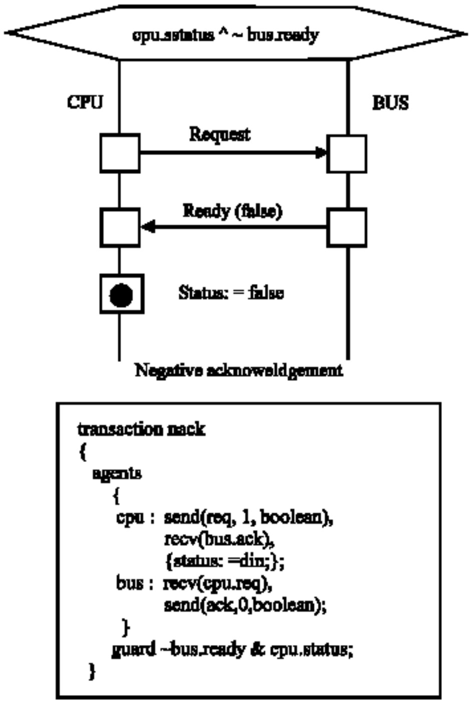
Fig. 4: Verdicts and counter example

and a transition occurs from _cpu4 to _cpu0. When the CPU sends a request or address to the BUS, it places that on the tail of the sending queue. When it wants to receive an acknowledgement or data it collects the acknowledgement or data from the receiving queue.

Properties in CTL and their verification: The SMV model checker is a self-contained C program as well as capable of reading an input file and accepting command line arguments. It has a line-oriented interactive mode, allowing each SMV invocation to check an arbitrary number of CTL formulas against a given specification.

Some properties of the bus protocols in CTL formula are verified with the SMV model checker. For clear illustration we have shown some of the properties as follows:

- SPEC AG (~_bus.ready-> AF (_bus.ready)) => if the bus is not ready once, eventually the bus will be ready. SMV shows the property is false.
- SPEC AG (~_cpu.status-> EF (_cpu.status)) => if the CPU status is false, eventually the status will be true in at least one node. SMV shows the property is true.
- SPEC AG (_bus.ready and ~_bus.empty_cpu_bus -> AX (_cpu.status and ~_cpu.empty_bus_cpu)) => once the BUS is ready to send data when the queue is not empty; in all the next states the CPU receives the data when the acknowledgement queue is not empty. The SMV shows the property is false. The properly verification results are shown in Fig. 4.



```
transaction nack
{
  agents
    {
     cpu :  send(req, 1, boolean),
            recv(bus.ack),
            {status: =din;};
     bus :  recv(cpu.req),
            send(ack,0,boolean);
    }
    guard ~bus.ready & cpu.status;
 }
```

Fig. 5: Modeling a system by TSL from MSC

PERFORMANCE ANALYSES AND COMPARATIVE STUDY

Now we will discusses about some of the major achievements TSL have gained in specifying a system behavior. We have used Line of Code Analysis for a comparative look over the performance of the TSL and SMV. For 4GT language LOC analyses is not an ideal measure. But as there is no visual tool for characterizing the system specification and one have to code the specification manually we took this measure for granted.

Easy modeling a system behavior: As we have seen in the examples in the study, the system specification of the simple bus protocol is much easier to characterize by TSL than SMV.

The code of TSL is much more similar to the system behavior as it depicts only the transactions of the communicating process, where as the SMV depicts all the states and state transition of a system, which is not so easy to characterize.

For clear illustration lets have a look in the example of Fig. 5. The negative acknowledged MSC in the example of study, have three actions under CPU process and 2 actions under BUS process. The TSL code segment for this MSC is stated beside. Comparing the two systems we can easily understand the simplicity of TSL to characterize a system behavior from MSC's. Even for a naïve user of TSL can model a system in TSL from MSC's.

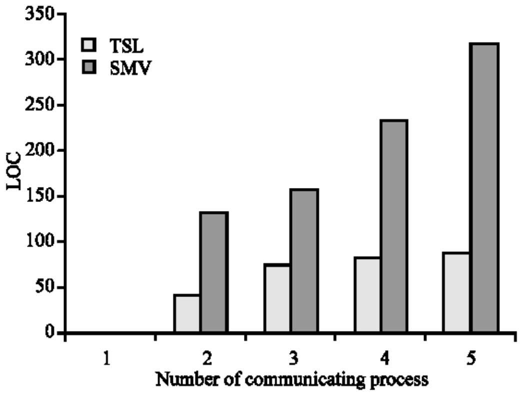


Fig. 6: Comparision between TSL and SMV

Reduction of Line of Code (LOC): A significant improvement of the effort, need to characterize the system, has been achieved by TSL. We calculate the improvement by Line of Codes (LOC) analysis. We saw to characterize the simple bus protocol in the example in the study, TSL needs 41 lines to code where as SMV need approximately 130 lines to code the same. So the reduction of LOC is more then 55%. But if we increase the number of transactions between the numbers of processes of the system, the improvement ratio is not so high. Still the improvement is very much significant. We have examined this phenomenon with some input data i.e. with some system characterization by TSL and SMV and plot the required LOC on the function of number of processes. The result the experiment is shown in Fig. 6.

CONCLUSION

In this study, we have shown only the verification of the Bus Protocols with our proposed system. We can verify all the communicating protocols in the same way. By TSL, with the help of MSC's, we can specify all the behaviors of the interacting components in the whole communication domain.

At present the transaction set i.e. the schemes modeled in the TSL form should be sequential depending on transactions. TSL fail to characterize the non-communicating system specification. For modeling a system by TSL without MSC's, requires expert user.

Though TSL needs MSC's for better performance, our observation is that we can model a whole system with TSL without the use of MSC's. Thus we can omit the all-informal procedures to automate the Formal Verification process. Our future concern is to establish TSL as a stand-alone modeling language and to apply our system in communication and embedded system domain.

REFERENCES

Aloul, F. and K. Sakallah, 2000. Efficient verification of the PCI local bus using Boolean satisfiability. In proceedings of International Workshop on Logic Synthesis (IWLS), http://eecs.umich.edu.

Holt, A., 1999. Formal verification with natural language specifications: guidelines, experiments and lessons so far. South Afr. Comput. J., 24: 53-257.

Halder, N., L. Ahmed and M. Asaduzzaman, 2005. Modeling and Formal Verification of Communication Protocols for RPC. B.Sc. Thesis. Computer Science and Engineering Discipline, Khulna University, Bangladesh.

Mokkedem, A., R. Hosabettu, M. Jones and G. Gopalkrishnan, 2000. Formalization and analysis of a solution to the PCI 2.1 bus transaction ordering problem. Formal Methods Syst. Des. Arch., 16 (1): 93-119.

Rudolpha, E., J. Grabowski and P. Graubmann, 2008. Tutorial on Message Sequence Charts MSC'96. www.rennes.supelec.fr/ren/perso/ bjouga/documents /sdl_msc/biblio/msc96tutorial.pdf.

Talukder, K.H., 2003. An Introduction to Formal Verification Based on SMV, in souvenir of National Seminar on Computer And Information Technology, Khulna University, pp: 66-71.

Talukder, K.H., 2003. Formal verification of the alternating Bit Protocol. In: Proceedings of International Conference on Information and Technology (ICCIT), Dhaka, Bangladesh, pp: 875-879.