Journal of Applied Sciences

ISSN 1812-5654

Journal of Applied Sciences 6 (8): 1712-1719, 2006
ISSN 1812-5654
© 2006 Asian Network for Scientific Information

Formal Specification and Validation of Selective Acknowledgement Protocol using Z/EVES Theorem Prover

Zarina Shukur, Nursyahidah Alias, Mohd Hazali Mohamed Halip and Bahari Idrus
Fakulti Teknologi Dan Sains Maklumat, Universiti Kebangsaan Malaysia
43600 Bangi, Selangor, Malaysia

Abstract: Selective ACKnowledgment (SACK) is a complex communication protocol as it is used in various types of distributed computer systems and networks. This acknowledgment mechanism is used with sliding window protocol that allows the receiver to acknowledge packets received out of order, but within the correct sliding window. In this study the SACK protocol has been specified by using Z formal specification language. A formal Z specification provides validation function to ensure that the specification is complete and consistent. The completeness and consistency of Z specification can be checked by proving initial state theorem, pre-condition and properties of a system specification. This study demonstrates the validation process of Z specification of SACK by using theorem proving technique. A theorem prover tool called Z/EVES is used to support the process. It helps to reduce time, energy and mistake than in relatively manual theorem proving which can be tedious and error-prone task.

Key words: Formal specification, protocol communication, safety property, formal validation

INTRODUCTION

Validation refers to a set of activities, which ensure that a developed product conforms to its specification. Barden *et al.* (1999). Define validation as a process of checking whether a specification is correct to the real-world condition. For example, formal specification satisfies a system requirement or that implementation is acceptable to the customers and validation needs to show that properties of the specification are real-world properties. As in (Jacky, 1997), it is necessary to show that a formal specification expresses the intent of the prose requirements. There are several ways to validate a system. These include Formal Technical Review (FTR) technique, viewpoint resolution technique, animation, symbolic execution and theorem proving. Within these techniques, theorem proving is the most reliable technique.

Formal proving is a complete argument of mathematical representation and it is used to validate statement about system description. This shows that formal proving can be used as one of the validation techniques. Formal proving can be done manually or with the support of formal method tools (WetStone Technologies, 1999) such as theorem prover tools (Wing, 1990) such as proof checker (Azurat, 2002). Theorem prover is a tool that implements automatic theorem proving without the need of user support (WetStone Technologies, 1999). Manual proving using humans is a long and looping process and there is a great possibility a mistake will be made. Therefore in order for humans to check proofs efficiently the proofs should not be unreasonably large and they should be presented in a user-friendly fashion. However, much of the proof involved in software verification is naturally detailed, low-level and repetitious and often results in large proofs-in short it is unsuitable for human checking. Thus, formal proving supported by tool may not only reduce the possibility of mistake but not totally removes it (Bowen and Hinchey, 1995). Therefore, the use of support tool is a main factor that can effect the acceptance of formal method practically (Babich and Deotto, 2002). In this study, one theorem-proving tool is chosen to support formal proving process, which is Z/EVES (Meisels and Saaltink, 1997). Z/EVES is chosen as a support tool because it can be applied in most process and need only a minimum background education such as Degree to use it. It can be learned in a few months depending on the type of applications and can be run in many platforms such as Unix, SunOS, Linux and Windows (Nursyahidah *et al*., 2004). The details about Z/EVES can be referred in (Meisels and Saaltink, 1997) and (Saaltink, 1997).

Formal proving as a validation technique is used to ensure specification has properties as it is required (Babich and Deotto, 2002). Formal validation can only be

Corresponding Author: Zarina Shukur, Fakulti Teknologi Dan Sains Maklumat, Universiti Kebangsaan Malaysia, 43600 Bangi, Selangor, Malaysia Tel: 03-8921-6720 Fax: 03-89216184

done onto the specification that was developed using formal method. In brief, formal method refers to the use of technique from formal logic and mathematic (NASA, 2002) in the specification, development, verification and validation phase with particular objectives (Giunchiglia and Traverso, 2000). In this study, the chosen formal specification to be developed is the specification for Selective ACKnowledgment (SACK) which is a part of TCP (Transmission Control Protocol) communication protocol.

Communication protocol is a complex protocol and it is used in many distributed system and networks. Non-formal techniques are successfully used to design the protocol, but it contains unexpected error and unwanted behavior (Bochmann and Sunshine, 1983). Validation needs to be done on a formal specification so that the unexpected error and unwanted behavior are discovered in the earlier development phase in order to design the correct protocol.

One of the formal validation techniques that involve formal specification is to prove an initial state, precondition and properties aspects (Barden *et al.*, 1999). The three aspects are proved by developing theorem and then proving process is executed to prove these theorems. If these theorems cannot be proved, it means that the Z specification was not consistent. Therefore, the specification needs to be reviewed and corrected. After that, the proving process is done once again. This shows proving is a repetitious process until the theorem can be proved.

OVERVIEW OF SACK

Transmission Control Protocol (TCP) acknowledgement system is a reliable sliding window transport protocol and flow control mechanism used on the Internet today. Selective ACKnowledgement (SACK) is a newer mechanism that allows the receiver to let the sender know what packet has been received. SACK is used to report multiple lost segments. In SACK, a window of TCP segment may be sent and received before an acknowledgement is received by the sender. Sender, receiver and channel are the main basic components in SACK mechanism. The flow control works when the sender first receives a message from the user application process. The message is then put in the transmission's buffer at the sender. The message is segmented and a unique sequence number is attached to every segment before it is sent to the receiver through a channel. The receiver buffer the received data segment at the receiver. To validate the received segments, the receiver transmits ACK to the sender with a sequence number for the next segment it is waiting to receive. If the receiver does not receive the data segment, it will asks the sender to make a retransmission of that segment. This provides reliability, as the sender retransmits any segments that are not acknowledged by the receiver. Smith and Ramakrishnan (2002) model the components, which consist of a sender, a receiver and two channels using I/O automata method as shown in Fig. 1. Basic structure for the formal model described in the figure includes a sender, a receiver, a channel for packets from the sender to the receiver and a channel for packets from the receiver to the sender. All these component is presented by the symbol S, R, C_{sr} dan C_{rs}.

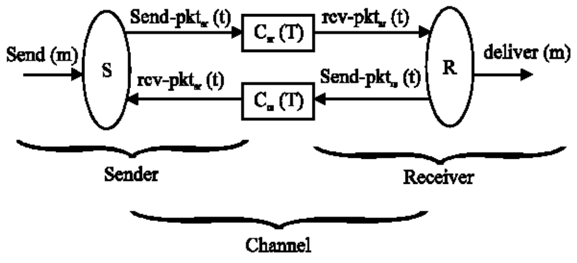

Fig. 1: Structure of the SACK formal model shows the four basic components

SACK Sender: Figure 1 shows that the sender has two input operations which are send(m) and rcv-pkt(t). The figure also shows that the sender has one output operation that is send-pkt(t). However, in detail implementation of SACK mechanism, (Smith and Ramakrishnan, 2002) identify three input operations, three internal operations and one output operation for the sender. Thus, there are seven operations for the sender as describe below:

- Input operation that receives message from user application, send(m)
- Input operation that receives validation from the receiver upon reception of the data segment, rcv-pkt(t).
- Input operation that receives validation from the receiver upon reception of the data segment and ask for a retransmission of a data segment, rcv-pkt(ack, b1, b2, b3).
- Internal operation that prepares a data segment to be sent to the receiver, prepare-new-seg(s).
- Internal operation that prepares a retransmission data segment to be sent to the receiver, prepare-retran-seg(s).

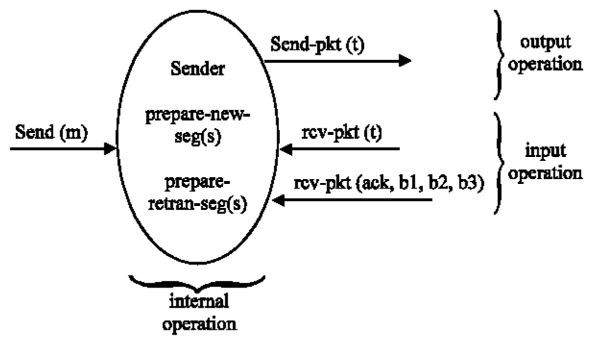

Fig. 2: Input, internal and output operation for sender

- Internal operation that causes a state of a data segment in retransmission buffer to be set to not yet received by the receiver. This operation is enabled if time for retransmission is expired, reset-sack.
- Output operation that sends a data segment to the receiver, send-pkt(t).

All the seven operations are presented in Fig. 2.

Z SPECIFICATION OF SACK'S SENDER

Z specification of SACK sender declares variables that are used in the sender automata model into a form of paragraphs. State, initial state and operation of the sender is declared into a number of schemas. There are seven operation schemas in the Z specification for SACK sender: *send*, *prepareNewSeg*, *sendPkt*, *rcvPkt*, *rcvPkt1*, *prepareRetranSeg* and *resetSack*. The following section presents the specification.

Global variables: Sender has a global variable which presents a message, sequence number of a data and some other variables. This section discuss on developing a Z specification for the sender variable. The message received from a user is hold into a buffer. The buffer is presented as a *BYTE* variable value 0 or 1 as follows:

BYTE :: = zero|one

The message is segmented into several segments. The size of a segment is a constant value and is presented as a *MSS* variable as follows:

MSS = = 536

In this specification, we assume that one segment can have a size of an alphabet data. A *ByteInt* variable represents a data segment. Each segment is presented by Byte variable and a sequence number of the segment is presented by *Seqnum* variable. The *ByteInt* variable is declared as follows:

ByteInt = = BYTE×Seqnum

Value of *Seqnum* is based on the value of WS, which is a window size in the sender. The window size is fixed with a constant value of 8. Thus, a value of *Seqnum* is between 0 and 7. Both of the variables are presented as follows:

WS = = 8
Seqnum = = 0 .. 7

Sender also has a retransmission buffer which contains data segments, sequence number of the data segment and a state of the data segment. This is presented by *SByte* variable as follows:

Sbyte == BYTE×Seqnum×BOOL

BOOL variable shows the state of a data segment, either the data segment has been received or not by the receiver. If the data segment is received by the receiver, the value of the *BOOL* is set to *TRUE*. Otherwise, the value is set to *FALSE* as follows:

BOOL :: = TRUE|FALSE

Blk variable shows a sequence number of a data segment place on the left and right of a sequence number of retransmission data segment and is presented as follows:

Blk = = Left×Right

State of the sender: State for the sender is represented by variables called state variables. The state variables are as follows:

- *sendBuf* represents a sender buffer which contain messages sent from user application. The type of the *sendBuf* variable is seq *BYTE* and this variable shows that the messages in the buffer are in a sequence as it was sent from the user application.
- *segmen* presents a segmented messages and its type is seq *ByteInt*. The variable shows the segment is in a sequence as in the sender buffer.
- *retranBuf* variable represents a retransmission buffer and its type is *SByte*. The variable shows a data segment and its state (whether it was received by the receiver). The retransmission buffer is used when the receiver does not receive a data segment and therefore, a retransmission is needed.
- *readyToSend* represents a state of the sender whether it is ready to send data or not.

- *sndUna* represents the sequence number of data segment which is not yet validated its reception by the receiver.
- *sndNxt* represents the next sequence number of data segment to be transmitted.

All these variables are declared in state schema as follows:

```
── Sender ──────────────────────────
sendBuf: seq BYTE
retranBuf: seq SByte
segmen: seq ByteInt
ready ToSend: BOOL
sndUna: Seqnum
sndNxt: Seqnum
────────────────────────────────────
```

In order to simplify the mathematical statements in the specification, five auxiliary variables are introduced, as in schema originalMessage. These variables are used to store the original information of the respective message.

```
── OriginalMessage ─────────────────
message: seq BYTE
senderData: seq BYTE
receiverData: seq BYTE
retransmitData: seq BYTE
dataSegmen: seq BYTE
────────────────────────────────────
```

Initial state of the sender: An initial state for the sender needs to be declared by identifying the initial value for every variable in the state schema. Initial value for the buffer, retransmission buffer and segment is an empty set. This shows there is no data has been received and transmitted by the sender. The sender too is in not ready state to sent any data. The value of the sequence number of received data segment which has not yet been validated by receiver and also the next sequence number of data segment need to be transmitted is set to 0.

The overall init state schema for the sender is presented in schema *InitSender*.

```
── InitSender ──────────────────────
Sender′
──────────────
sendBuf′ = ⟨⟩
retranBuf′ = ⟨⟩
segmen′ = ⟨⟩
readyToSend′ = FALSE
sndUna′ = 0
sndNxt′ = 0
────────────────────────────────────
```

Operations of SACK' sender: All the schema causes changes to state schema of the sender which are presented by expression Δ*Sender*. An explanation about all the schemas is as follows:

***send* schema:** *send* schema is an input operation for the sender. The sender receives message from the user application. Expression *m?* represent the message. The message is hold at the sender buffer. The expression has ' symbol that shows the variable state after an operation is executed. The *send* operation schema is as follows:

```
── send ────────────────────────────
ΔSender
ΔOriginalMessage
m?: seq BYTE
──────────────
sendBuf' = sendBuf^m?
retranBuf' = retranBuf
segmen' = segmen
readyToSend' = readyToSend
sndUna' = sndUna
sndNxt' = sndNxt
message' = m?
senderData' = sendBuf'
receiverData' = receiverData
retransmitData' = retransmitData
dataSegmen' = dataSegmen
────────────────────────────────────
```

***prepareNewSeg* schema:** *prepareNewSeg* schema is an internal operation for the sender. It shows the sender prepares a segment that is needed to be transmitted to the receiver. This operation will only be executed if the state of the sender is in not ready position to send any of the segment, the buffer has a data and the next sequence number of data segment to be transmitted is less than the size of the window. The data, which is at the front of the buffer, is taken out from the buffer to be segmented and it is assigned with a sequence number. The segment is grouped with the state of the data segment which is not yet received by receiver and hold it at the retransmission buffer. After segmentation, the state of the sender is in ready to send the data segment. The *prepareNewSeg* operation schema is as follows:

```
── prepareNewSeg ───────────────────
ΔSender
ΔOriginalMessage
──────────────
readyToSend = FALSE
sendBuf ≠ ⟨⟩
sendNxt< sndUna + WS
if sndNxt + 1>7 then sndNxt' = 0 else sndNxt' = sndNxt+1
sendBuf' = tail sendBuf
segmen' = segmen^ ⟨(head sendBuf, sndNxt)⟩
readyToSend' = TRUE
sndUna' = sndUna
retransmitData' = retransmitData^⟨head sendBuf⟩
```

```
dataSegmen' = dataSegmen⁀⟨head sendBuf⟩
receiverData' = receiverData
senderData' = senderData
message' = message
```

***sendPkt* schema:** *sendPkt* schema is an output operation for the sender. In this operation, the sender transmits the data segment to the receiver. This operation will only be executed if there is a data segment to be sent and the state of the sender is in ready to send data segment. After the segment has been transmitted, state of the sender is in not ready to send a data segment. The *sendPkt* operation schema is as follows:

```
sendPkt
ΔSender
seg?: seq ByteInt
────────
seg? = segmen
readyToSend' = TRUE
sendBuf' = sendBuf
segmen' = segmen
retranBuf' = retranBuf
sndUna' = sndUna
sndNxt' = sndNxt
```

***rcvPkt* schema:** *rcvPkt* schema is an input operation for the sender. It shows the sender has received a validated sequence number that was received by the receiver. This operation receives the next sequence number of data segment need to be transmitted. This operation will only be executed if an expression *sndUna<ack?<sndNxt* is true. If this is true, it shows that the segment which has a sequence number before the value of *ack?* has been received successfully by the receiver. Thus, the segment will then be taken out from the retransmission buffer and the value of the next sequence number of data segment need to be transmitted is being updated. The *rcvPkt* operation schema is as follows:

```
rcvPkt
ΔSender
ΔOriginalMessage
ack?: Seqnum
────────
retranBuf ≠ ◊
if sndUna < ack? ≤SndNxt
then retranBuf' = tail retranBuf ∧ retransmitData' = tail
retransmitData
else retranBuf' = retranBuf ∧ retransmitData' =
retransmitData
readyToSend' = readyToSend
sendBuf' = sendBuf
segmen' = segmen
sndUna' = ack?
receiverData' = receiverData
senderData' = senderData
message' = message
dataSegmen' = dataSegmen
sndNxt' = sndNxt
```

***rcvPkt1* schema:** *rcvPkt1* schema is an input operation for the sender. It shows that the sender receives a validated sequence number that was received by the receiver and need to be retransmitted. This operation receives the next sequence number of data segment need to be transmitted and the sequence number of data segment to be retransmitted depends on a value of *b1?, b2?* and *b3?*. This operation will only be executed if the expression *sndUna<ack?<sndNxt* is true. If this is true, it shows that the segment which has a sequence number before value of *ack?* has been received successfully by the receiver. So, the segment will then be taken out from the retransmission buffer and the value of the next sequence number of data segment to be transmitted is being updated. Next, every sequence number in retransmission buffer is being checked in order to determine which packet needs to be retransmitted. The checking is made using the following expression:

- If the sequence number of the segment in the retransmission buffer<*b2?*, or the sequence number of the segment in the retransmission buffer>*b3?*, then the state of the segment is as in the original state.
- If *b2?*<the sequence number of the segment in the retransmission buffer<*b3?*, then the state of the segment is that the segment has been received by the receiver.

The *rcvPkt1* operation schema is as follows:

```
rcvPkt1
ΔSender
ΔOriginalMessage
ack?: Seqnum
b1?: Seqnum
b2?: Seqnum
b3?: Seqnum
────────
retranBuf ≠ ◊
retransmitData ≠ ◊
if sndUna < ack? ≤ sndNxt
then retranBuf' = tail retranBuf
  ∧ retransmitData' = tail retransmitData
  ∧ (∀ e: ran retranBuf
     • (if e.2<b2?∨e.2≥b3? then e.3 = e.3 else e.3 = e.3)
       ∨ (if b2?<3.2≤b3? then e.3 = TRUE else e.3 = e.3)))
else retranBuf' = retranBuf
readyToSend' = readyToSend
sendBuf' = sendBuf
segmen' = segmen
sndUna' = ack?
receiverData' = receiverData
senderData' = senderData
message' = message
dataSegmen' = dataSegmen
sndNxt' = sndNxt
```

***prepareRetranSeg* schema:** *rcvPkt1* schema is an internal operation for the sender. It shows that the sender is preparing a segment that need to be retransmitted to the receiver. This operation will only be executed if state of the sender is in not ready to send a data segment and the retransmission buffer has a data. The data segment which was not received by the receiver in the retransmission buffer is then segmented. Next, the state of the sender is in ready to send a data segment. The *prepareRetranSeg* operation schema is as follows:

```
┌─prepareRetranSeg ───────────────
ΔSender
ΔOriginalMessage
├──────────
readyToSend = FALSE
retranBuf ≠ ◊
∀ e: ran retranBuf; f: segmen
  • if e.3 = FALSE
    then segmen' = segmen ^ ⟨(e.1, e.2)⟩
    else segment' = segment
readyToSend' = TRUE
sendBuf' = sendBuf
retranBuf' = retranBuf
sndUna' = sndUna
sndNxt' = sndNxt
receiverData' = receiverData
senderData' = senderData
retransmitData' = retransmitData
message' = message
dataSegmen' = dataSegmen
└─────────────────────────────────
```

***resetSack* schema:** *resetSack* schema is an internal operation for the sender. It shows that all of the segment state in the retransmission buffer is not yet received by the receiver. This operation will only be executed if time for retransmission is expired. The *resetSack* operation schema is as follows:

```
┌─resetSack ───────────────────────
ΔSender
├──────────
∀ e: ran retranBuf'
  • ∀f: ran retranBuf
    • e.2 ∈ Seqnum ∧ f.2 ∈ Seqnum ∧ e.3 = FALSE ∧
        e.1 = f.1 ∧ e.2 = f.2
segment' = segmen
readyToSend' = readyToSend
sendBuf' = sendBuf
sndUna' = sndUna
sndNxt' = 0
└─────────────────────────────────
```

FORMAL VALIDATION OF Z SPECIFICATION OF SELECTIVE ACKNOWLEDGEMENT PROTOCOL

The methodology of formal validation of Z specification for SACK sender is shown in a flow chart as in Fig. 3.

The methodology of formal validation of Z specification for SACK sender starts with the translation of the I/O automata model of protocol SACK (Smith and Ramakrishnan, 2002) to the Z specification in LaTEX format. Then, the Z specification is type checked. Type checking is a condition before using a theorem proving function in Z/EVES (Nursyahidah and Zarina, 2006). Type checking is used to check if the specification has a type checked error. If there is no type error, next is to develop a initial state, pre-condition and properties theorem for proving in the same Z specification. Type checking is also done to these theorems to ensure there is no type error. If there is no error, then next is to prove the three theorems. Type checking and theorem proving process need to be repeated until the theorems is proved correct and produced a complete and consistent specification.

Type checking: The Z specification of SACK sender, which was written in LaTEX format, has been type-checked. Type checking must be done as a condition before using a theorem proving function in Z/EVES theorem prover. Type checking is used to check if the specification is free from type error. An example of an equation predicate expression contains type error is as follows:

$$sendBuf = zero$$

In the equation predicate above, a variable in the right side must have the same type with a variable in the left side. The above expression is type-checked using Z/EVES and produces error message as follows:

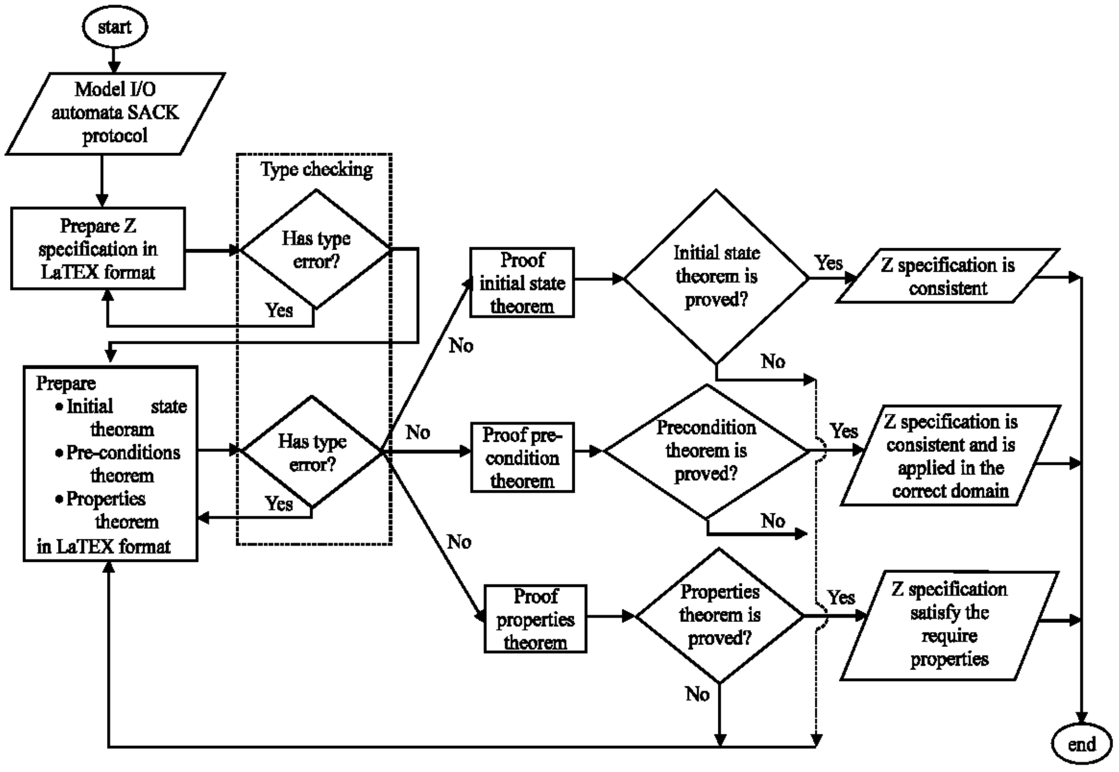
start
Model I/O automata SACK protocol
Prepare Z specification in LaTEX format
Type checking
Has type error?
Yes
Prepare
• Initial state theoram
• Pre-conditions theorem
• Properties theorem
in LaTEX format
Has type error?
Yes
No
No
No
Proof initial state theorem
Proof pre-condition theorem
Proof properties theorem
Initial state theorem is proved?
Precondition theorem is proved?
Properties theorem is proved?
Yes
No
Yes
No
Yes
No
Z specification is consistent
Z specification is consistent and is applied in the correct domain
Z specification satisfy the require properties
end

Fig. 3: The methodology of formal validation of Z specification for SACK sender

Error TypesNotSame (line 71) [Type checker]: types of \Local sendBuf and\Global zero are not the same.

The above error message shows that in Line 71, the declared *sendBuf* and *zero* in the equation predicate *sendBuf* = *zero* does not have the same type.

Proving of initial state theorem: An intial state theorem is used to show that at least one condition for SACK does exists. A simple state to show and to prove is the initial state (Woodcock and Davies, 1996). One initial state theorem has been developed for Sender of SACK based on it's initial state schema. The following shows the initial state theorem (Nursyahidah and Zarina, 2006).

theorem *TheInitSender*
$\exists$ *Sender' • InitSender*

This theorem has been proved by only using command proving that is a prove by reduce command.

Proving of precondition theorem: The use of proving of precondition theorem is to show every operation is not applied beyond the domain, which is a situation where the output from the operation is not recognized (Woodcock and Davies, 1996). This can be shown with the state before of a particular operation, exactly relates at least to the one of the state after the operation. It means that the state before an operation and after the operation clearly satisfy the specified relation (Spivey, 1998).

To demonstrate how the proving of precondition is done, an operation of receiving packet by a sender will be used as an example. This operation receives the next sequence number of data segment that is needed to be transmitted. This operation will be executed if the number is greater than the value of *sndUna* and less than *sndNxt*. To check whether the specified operation can be applied within this domain, we develop the following theorem:

theorem *rcvPktPre*
$\forall$ *Sender; OriginalMessage; ack?: Seqnum; s: BYTE*
| *ack?* > *sndUna* $\wedge$ *ack?* $\leq$ *sndNxt* $\wedge$ *retranBuf* $\neq$ $\diamond$
• *pre rcvPkt*

Proving of safety properties: One of the two properties that are normally discussed in protocol communication are; safety. The other one is liveness. Safety properties are assertions that certain undesirable things do not happen (Duke and Rose, 2000). For example in communication protocol, the stream of messages received should be the same as the stream transmitted, without loss, replication or permutation. Based on the I/O automaton model by Smith and Ramakrishnan (2002), we develop several theorems that represent the safety properties of SACK protocol. These theorems are then proved by using Z/Eves theorem prover. In this section, we demonstrate two safety properties that can be validated by using Z/Eves.

First, safety property in sending operation; the message that is going to be sent, m, must be added at the back of sender buffer. The theorem that represents this statement is as follows:

theorem *OperationSend*
$\forall send; m?: seq\ BYTE \bullet sendBuf' = sendBuf ^\frown m?$

Second, safety property in preparing a new message; the message m, that is retrieved from the sender buffer (sendBuf) will be pushed ad the back of resending buffer, retranBuf.

theorem *OperationPrepareNewSeg*
$\forall prepareNewSeg \bullet retranBuf' = retranBuf ^\frown \langle (head\ sendBuf, sndNxt, FALSE) \rangle$

All of the theorems discussed in the above section have been proved by using Z/Eves theorem prover by using only one step command that is prove by reduce.

CONCLUSIONS

In this study, we present a Z specification of SACK (sender) which has been developed based on I/O automata model of (Smith and Ramakrishnan, 2002). Then, we demonstrate a validation process of Z formal specification of SACK sender using theorem proving technique. According to our experience, many theorems have been through a long and repetitious proving process. If the proving is done manually by humans, the possibility a mistake will be made is higher. With Z/EVES, not only this possibility can be reduced, the proving can be done fast and reliable.

REFERENCES

Azurat, A.I.S.W.B., 2002. A survey on embedding programming logics in theorem prover. (online) http://www.library.uu.nl/digiarchief/dip/dispute/2002-0308-131854/2002-007.pdf.

Babich, F. and L. Deotto, 2002. Formal methods for specification and analysis of communication protocols. IEEE Commun. Surveys and Tutorials, 4: 2-15.

Barden, R., S. Stepney and D. Cooper, 1999. Z In Practice. Great Britain. Prentice Hall International (UK) Limited.

Bochmann, G.V. and C.A. Sunshine, 1983. A survey of formal methods in computer network and architectures and protocols. IBM Corporation Yorktown Heights, New York. Plenum Press.

Bowen, J.P and M.G. Hinchey, 1995. Ten commandments of formal methods. Computer, IEEE, 28: 56-63.

Duke, R. and G. Rose, 2000. Formal object-oriented specification using object-Z. MacMillan Press Ltd.

Giunchiglia, F. and P. Traverso, 2000. Special section an theorem proving, Theorem proving in technology transfer: the user's point of view. Intl. J. STTT 3: 1-12. Springer-Verlag.http://www.ora.on.ca/z-eves/welcome.html

Jacky, J., 1997. The Way of Z. United States of America. Press Syndicate of the University of Cambridge.

Meisels, I. and M. Saaltink, 1997. The Z/EVES Reference Manual (for Version 1.5). Ora Canada. Canada.

NASA ARC., 25 Januari 2002. V and V of Advanced Systems at Nasa. Task No: 10 TA-5.3.3 (WBS 1.4.4.5.3). Charles Pecheur, RIACS. Northrop Grumman Corp.

Nursyahidah, A., S. Zarina, I. Bahari and M.H. Mohd. Hazali, 2004. Pengesahsahihan Spesifikasi Formal Sistem Maklumat Pelajar Menggunakan Pembukti Teorem Z/EVES (Validation of Formal Specification of Student Information System Using Z/Eves Theorem Prover), in Prosiding Simposium Kebangsaan Sains Matematik ke Universiti Islam Malaysia, 12: 23-24.

Nursyahidah, A. and S. Zarina 2006. Experience of Validating the Sender of Selective Acknowledgement (SACK) Protocol By Using Z/EVES Theorem Prover. Technical Report. FTSM, Universiti Kebangsaan Malaysia.

Saaltink, M., 1997. The Z/EVES Mathematical Toolkit Version 2.2 for Z/EVES Version 1.5. Ora Canada. Canada

Smith, M.A. and K.K. Ramakrishnan, 2002. Formal specification and verification of safety and performance of TCP selective acknowledgment. IEEE/ACM Transaction On Networking, 10.

Spivey, J.M., 1998. The Z Notation: A Reference Manual 2nd Edn. England.

WetStone Technologies, Inc., 1999. Formal Methods Framework, F30602-99-C-0166, Final Monthly Status Report. Air Force Research Laboratory/IFGB, Rome, NY 13441-4505.

Wing, J.M., 1990. A Specifier's Introduction to Formal Methods. Computer. IEEE, 23: 8-23.

Woodcock, J. and J. Davies, 1996 Using Z: Specification, Refinement and Proof. Prentice Hall, London.