TR-82-012

A Simulator for Communication Protocols

By

Jyh-sheng Ke and Sheng-Ching Jeng

Institute of Information Science

Academia Sinica

This work was partially supported by National Council Grant NSC71-0404-E001-02

July 21, 1982

1. Introduction

The design of communication protocols for computer networks is one of the most challenging problems to the engineer. In the last few years a number of large scale networks have been implemented, and some of them have been illustrated in the literature. It has become clear that the design of the communication protocols for these networks is a difficult job. Many of the problems in designing these communication protocols, which involve asynchronous parallel processes, are due to the difficulty of realizing and analyzing the sequence of event occurrences.

For overcoming these problems, the major objective of our researches, we try to find out a good tool for the specification and verification of communication protocols. The subject of this report can be described as follows.

- Basing on Petri net theory to develop a model, which is called the Petri net Derived Model (PNDM), for the specification of communcation protocols.

- To develop a simulation system, which is called the Communication Protocol Modeling System (CPMS), for the specification and verification of communication protocols.

Part 2 is concerned about the Communication Protocol

Modeling System (CPMS) and Petri net Derived Model (PNDM). Part 3 explains the Protocol Specification Language (PSL). Part 4 explains the usage and implementation of CPMS simulator. Part 5 presents an example of constructing the alternating bit protocol and using the CPMS to simulate this protocol. In addition, appendices A and B list the general form and Backus Normal Form (BNF) of PSL, appendix C lists the specification of the alternating bits protocol. Appendix D is an example of CPMS simulation phase 2. And, appendix DR is an example of CPMS simulation phase 3.

2. The Communication Protocol Modeling System

Fig 2.1 shows the architecture of the Communication Protocol Modeling System , CPMS.

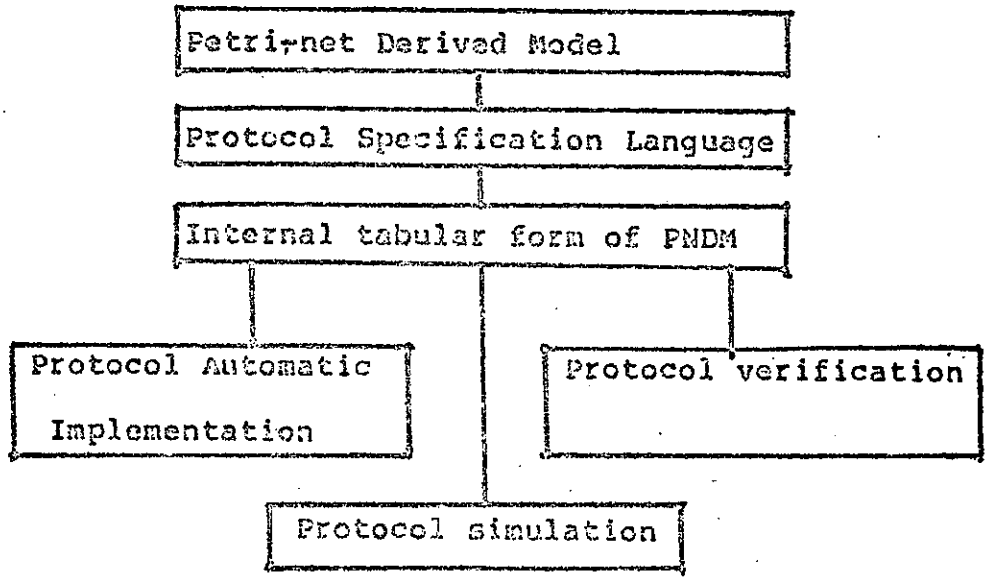


Fig-2.1 CPMS, Communication Protocol Modeling System

The designer of communication protocols must first use the Petri net derived model (PNDM) to describe the protocol which concerns what is needed to be done and how to do it. The PNDM is then transformed into the syntactic description by using protocol specification language, and is taken as the input of the CPMS through the operation of CPMS, i.e. through the simulation, to find out if any errors may take place. Repeatedly modify and simulate the PNDM until no more error can be detected. Consequently, protocol may be implemented semiautomatically.

2.1. Petri Net Derived Model (PNDM)

In this section, a new tool for protocol specification and verification, which is called Petri Net Derived Model, is described. PNDM is essentially an extended Petri net model.

The extensions of PNDM based on General Petri net may be categorized into four fields, including place extension, variable extension, transition extension, and function extension.

2.1.1. Place extension

In PNDM, a new type of places is introduced. The new type place is called external place in the sense that it has no input arc from any transition, but may generate tokens automatically when some special events occur. E.g, we may

use an external place to specify the condition that there is a new message needed to be transmitted from the sender of the communication system. When a new message needs to be transfered, the corresponding place will randomly automatically generate tokens in it.

2.1.2. Variable extension

Since the Petri net has no way to describe numerical events, we extend it by adding some variables.

This extension includes three types of variables, namely, structure variables, counter variables, and timer variables as follows. Each of these variables has its special usage and may increase the modeling power of Petri nets.

2.1.2.1. Structure variables

This type of variables is mainly needed for describing messages which are transfered between two communication stations. Usually, messages may have different formats, so we must provide some methods to describe them. To use structure variables, the user must declare their formats as follows.

variable name : simple type name (multiplicity) (1)

variable name : block type name (2)

Exapmle 2.1 : Message format

HEADER : B1(6) (3)

```
TITLE     : B16(1)                    (4)
DATA1     : B8()                      (5)
DATA2     : AX                        (6)
TAIL      : B1(6)                     (7)
```

Expressions (1) and (2) are formal forms of variable declaration. Type name is the name of data type which may be categorized into simple type and block type. Simple type includes three types : bit type, byte type, and word type, and their keywords are B1, B8, and B16, respectively. Block type is the combination of simple types, it is declared at the type declaration part. The multiplicity means the length of the variable with type declared in expression (1). At expressions (3), (4) and (7), the length of variables are static length with their multiplicity. And, at expression (5), B8() means the bytes string with dynamic length which is determined by the assignment statement explained at the transition extension part. In our CPMS simulator B8 has the same mean as B8(). At expression (5), AX is a name of block type which must be declared at the type declaration part.

The type declaration of PNDM is similar to the type declaration of PASCAL. The syntax of type declaration may be defined with BNF in appendix B. Here, we give an example of block type declaration.

Example 2.2 : Type declaration of PNDM

```
AX={DAX1=B16(2)                       (8)
```

DAX2=B8() (9)

DAX3=AXB (10)

DAX4=B16()} (11)

AXB={DAXB1=B16(2) (12)

DAXB2=B8()} (13)

We may declare type AX in expression (6) with expressions (8)~(13). In (8), DAX1 is the field name for reference. To refer the data declared at (8) or (12) with block type, we may use DATA2-DAX1 to refer data in (8) and DATA2-DAX3-DAXB1 to refer data in (12). That is, we add the dash '-' between the variable name and the field name, or between two field names, to construct a full name for variable reference.

In our CPMS, the structure variables are mainly used for message transfer. They may be assigned and compared. These two terms will be explained detailedly in the assignment-statement and if-statement of transition extensions.

2.1.2.2. Counter variable

This type of variable may be used for controlling the transition firing, i.e. it may influence the sequence of transition firing. The influential method will be introduced at the transition extension part.

The functions of counter variables include counter declaration, counter assignment, counter increasing, counter

modulus, and counter comparing.

Counter variables are declared at the variable declaration part, they must declare counter name with modulus value. For example, the counter name of A(8) is A and its modulus value is 8, i.e. when the value of counter A increases to 8, the value of counter will be set to be zero.

Each counter may be assigned by a value, and its value may be increased with one by increasing statement. The counter variables may also be compared in if statement.

2.1.2.3. Timer variable

Timer variable is similar to counter variable,and can be called as a decreasing counter.

The functions of timer variables include timer start statement, timer destroy statement, and timeout mechanism. The value of timer variables can be initiated by start statement in which value is the modulus declared at timer variable declaration.

The value of timer variable is automatically decreased by the execution of the simulator. If the value of timer variable decreases to zero, timeout mechanism of timer will be activated. And the timeout signal may influence the sequences of transition firing, because the condition of forcing function may be determined by timeout.

2.1.3. Transition extension

Transitions in PNDM may be divided into terminal transitions and node transitions. Terminal transitions are like transitions of Petri nets, but they may be described by transition action statements, and their firing actions about token flow may be determined by the values of counter variables. Node transitions may be replaced by a Petri net module supported by PNDM. In the following, these extensions will be explained.

2.1.3.1. Transition action description

The transitions of PNDM may use the action statements to describe the action of transition firing, and these action statements may be categorized into six types.

2.1.3.1.1. Counter statement

This is the operations about counter variables. Two types of counter statements are shown as follows :

..Counter assign statement

For example,

A <- 2

sets the counter variable have value 2.

..Counter increased statement

For example,

++A

means the value of counter variable A will be

incremented by one.

2.1.3.1.2. Timer statement

The timer statement describe the functions about the timer variables. Two types of timer statement are shown as follows :

..Timer start-statement

For example,

timer TA <- Ø

starts timer TA.

..Timer destroy-statement

For example,

<-timer TA

stops the mechanism of timer TA.

2.1.3.1.3. Transfer statement

These statements transmit or receive messages between two communication stations. Two types of transfer statement are shown as follows :

..Send statement

For example,

send 1 MESS

sends the message MESS with format declared at variable declaration and type declaration to transfer port 1.

..Receive statement

For example,

receive 1 MESS

receives the message MESS from transfer port 1.

When these two types of statement are executed and some errors are detected ,e.g. transmission error, they will be executed sequentially. If no error is detected, the execution sequence will skip one statement.

2.1.3.1.4. Terminate statement

In the transition action, the syetem may be stopped by these statements. Two types of terminate statement are shown as follows :

..Stop statement

For example,

stop

can stop the operation of simulator.

..Exit statement

For example,

exit

exits from the transition firing, and regard the transition as being fired.

2.1.3.1.5. Assignment statement

A structure variable may be assigned by this statement. For different types of variables, we must use different keywords for the simulator.

For example,

HEADER= %01001001 (14)

TITLE= 104 (15)

DATA1= "abcdef (16)

The equal symbol '=' represents assignment. And the percentage symbol '%' represents the header of bit string, and the double quote " represents the header of byte string, and the word variable is directly assigned by an integer.

In (14), the length of bit string assigned to HEADER is more than six, the over bits will be truncated, and the assigned bit string will be %010010. In (15), we assign value 104 to word variable TITLE. In (16), the variable DATA1 is a dynamic length byte string, and has been assigned with the value "abcdef".

In other situations, the length of string assigned may be less than the length of variable declared, we may augment bit 0, space, and zero, respectively, to the space of unassigned.

A special case may happen, i.e. the word string may be declared such that the assigned string is delimited by space, like TITLE2= 104 105

2.1.3.1.6. if-statement

The format of if-statement is :

if (comparing) statement-1 else statement-2,

If the condition of comparing is true, then execute statement-1, else execute statement-2. Statement-1 and

statement-2 may be any type of statement except if-statement. The comparing part may allow the counter-variable and non-counter-variable to be compared. The comparison of counter-variables, i.e. comparison between counter variables, which have integer values, includes five types of operators, namely, >, <, >=, <=, ==.

The comparison of non-counter variables, i.e. comparison between non-counter variables, which have values of the same format, only allows one type operator ==.

2.1.3.2 Transition controlled

This is the ability of transitions that they can select the path of token movings when a transition fires.

There may be two types of such transitions as shown in Fig 2.2 and Fig 2.3.

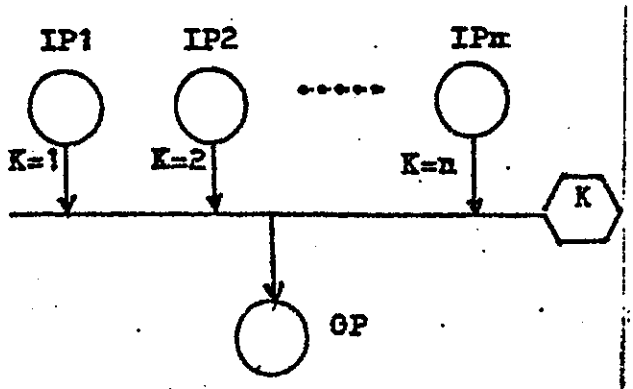


FIG 2.2

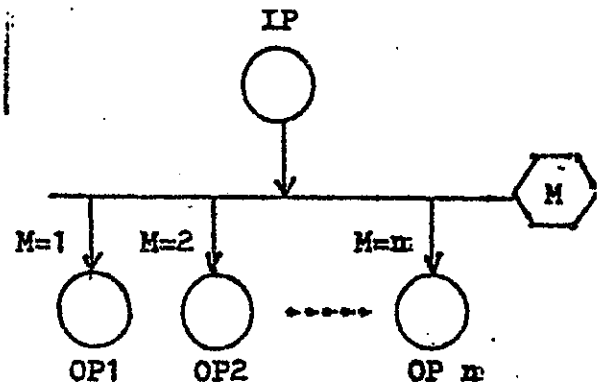


FIG 2.3

In Fig 2.2, the tokens moved are determined by the value of counter variable K. For example, if counter K has value one, tokens will be absorbed from place IP1 and sent to place OP. In Fig 2.3, the tokens moved are determined by the value of counter variable M, e.g. if counter M has value two, tokens will be absorbed from place IP and sent to the place OP2. Such functions must be represented by when-statement in transition action.

In Fig 2.2., it is represented as follows :

when(IP1*K==1:OP|IP2*K==2:OP|...|IPn*K==n:OP)

In Fig 2.3., it is represented as follows :

when(IP*M==1:OP1|IP*M==2:OP2|...|IP*M==n:OPn)

Text in front of the colon ':' is the condition which specifies if the place holds tokens or if the counter comparing is true, e.g. IP1*K==1, means that place IP1 holds one token and counter K has value 1. When condition matches, tokens will be sent to the place which is specified in text behind the colon, e.g. OP.

These extensions are similar to the macro E-net[NOE 73]. In fact, we borrow these concepts from E-net in order to enforce the controlled power of Petri nets.

2.1.3.3. Transition hierarchy

Transition hierarchy is an extended property of transitions. The transition of PNDM, which is called sub-transition, may be replaced by a Petri net structure.

This property is good for analysis, we can make each sub-transition correctly, and then verify the correctness of the whole model by regarding each sub-transition as a general transition.

2.1.4. Function extension

PNDM allows users to set the initial value of variables, and to terminate the PNDM by describing the termination condition. It also allows the force function to change states of PNDM.

2.1.4.1. Initial function

The users may initiate all places and counter variables by using initial function at the module initialization part. For example, p1:2 means that place p1 holds two tokens, and A <-2 means counter A is initiated to have value two.

2.1.4.2. Termination function

The users can specify the normal termination condition of PNDM at the module termination part. While the status of PNDM matches the terminated condition described by the user, the simulator will terminate and restart itself to simulate again.

The termination condition is composed of places names with the number of tokens they hold. For example, p1:2; p2:1; ,i.e. place p1 holds two tokens , place p2 holds one token, and no more tokens exist in other places.

2.1.4.3. Forcing function

This function is similar to the termination function, except that it will not be terminated. The uesr can describe many forcing functions and each includes two parts, one is forcing condition, and the other is forcing action.

For example,
```
p1:2; p2:4;
timeout t01;
force
p1:1; p2:1; p3:1;
```

At this forcing function, texts in front of "force" are the conditions, and behind it are the forcing actions. While place p1 holds two tokens, place p2 holds four tokens, and timer t01 is timeout, i.e. the forcing condition matches, the PNDM will be enforced into the forcing action, i.e. each of place p1, p2 and p3 holds one token.

Using forcing function can increase the modeling power of PNDM, particularly we can add the timeout mechanism into the forcing condition, hence the system modeled may be more close to the real world.

3. Protocol Specification Language (PSL)

For the sake of computer processing, the protocol designer must transform the PNDM into the protocol specification language (PSL). Appendix A lists the general form of PSL and appendix B lists the BNF of PSL. At appendix A, the

capital letters are the keywords in PSL, for convenience, we replace them by shorter keywords which are in parentheses at the same line. We use the three dots to represent the repetition which is the same as the last text line. And, the protocol specification language is used for system modeling, its basic unit is called module, which may include eight parts.

Place declaration and transition declaration are used to declare the places and transitions by their names. These two parts must exist in every module.

The data types used for variable declaration are declared at the type declaration part. At the variable declaration part , there may be three types of variable declaration, structure declaration declares the data variables for message transfer, counter declaration declares counter variables, and timer declaration declares the timer variables. The system is initiated at the module initialization part which is one type of function extensions, and is explained in 2.1.4.1. At this part, the designer may initiate the places with the number of tokens held and the value of counters which have been declared.

Next part is called the module termination part, the designer can specify the termination conditions which may be more than one, and each termination condition may be represented by a place with the number of tokens to be held.

Another type of function extension is called forcing function which is explained in 2.1.4.3., also can be transformed into the module forcing part with PSL. It is similar to module termination except that the forcing function doesn't terminate while the condition matches, and will go into a new situation defined by force part of forcing function. Its condition may be determined by the timeout of timer.

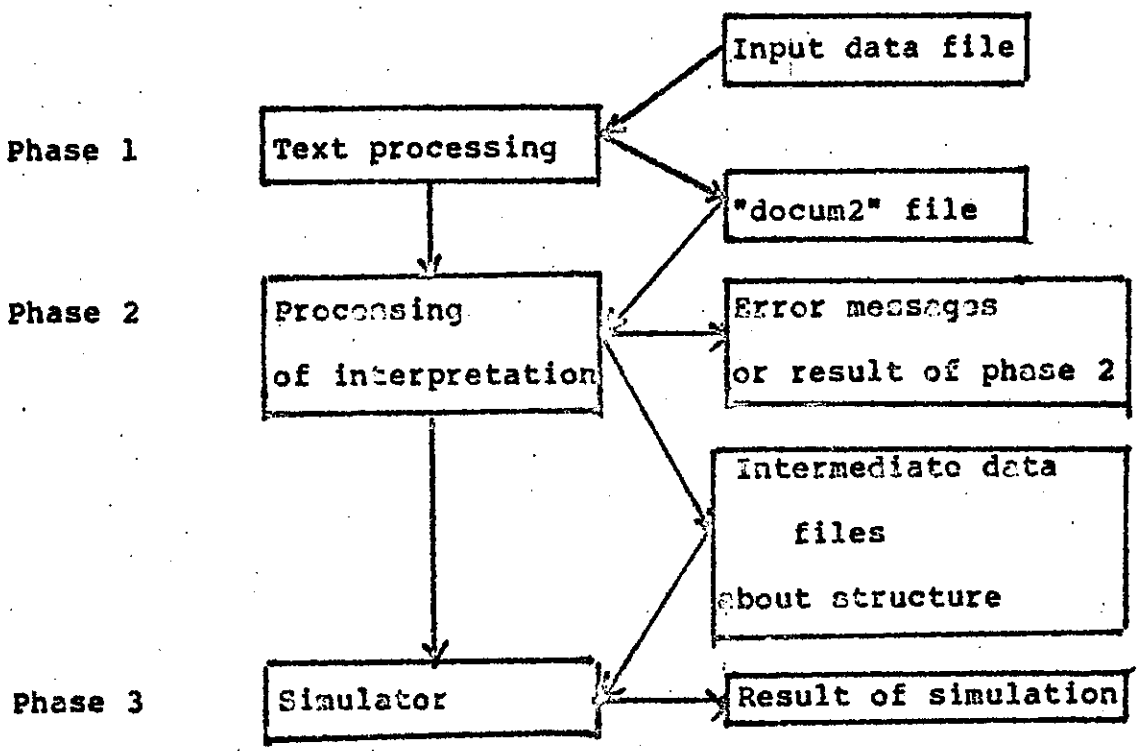
The last part is the most important in PSL, mainly defines the structure of Derived Petri Net. It is called transition description part. Each transition structure must be described in this part and include the input places and output places and transition action explained in 2.1.3.1 or a sub-transition described by a module explained in 2.1.3.3.

The place declaration and transition declaration must exist in every module. However the other six parts are optional.

In summary, after the model described by PNDM is transformed into the language represented by PSL, the designer can use it as the input of the CPMS simulator.

4. Implementation and Usage of CPMS simulator

4.1. Implementation of CPMS simulator



The implementation of CPMS simulator is divided into three phases. Text processing is the work to be done at phase 1. In phase 1, the input data file is properly formatted into the output data file with file name "docum2" and it is used as the input of phase 2.

At phase 2, its work mainly is checking and interpreting the structure of the system modeled. If errors exist, the error messages will be provided by the simulator. The simulator will produce the intermediate data files for being used by phase 3. These data files include places and

transitions declaration files, named "pfile" and "tfile", counter, structure and timer declaration files named "vcfile", "yifile" and "vtfile" , termination and forcing data files named "termifile" and "forcefile", data type file named "strfile", and transition data files each of which corresponds to one transition which is called "t****".

At phase 3, the simulator uses information of the intermodiate data files provided by phase 2 to simulate the modeled system.

The simulator first initializes all variables and prompts questions about simulation ranges, then the simulator simulates the modeled system, i.e. the communication protocol, specified by the designer.

4.2.Using CPMS to simulate the communication protocol

In this section, we introduce the method of using the CPMS to simulate the communication protocol.

The CPMS is a modeling system for communication protocols. When the protocol designer wants to use the CPMS to design a protocol, he first must use the Petri net derived model (PNDM) to describe the protocol, and then transform the PNDM into the protocol specification language (PSL). Next, use the PSL as input of CPMS. Through CPMS processes, the designer will know if any errors taking place in the communication protocol, and repeatedly modify the PNDM until

no error can be detected.

To use CPMS, the protocol designer first edits a data file which contains the PSL specification of the modeled system. This data file will be used as input of the CPMS simulator. The CPMS simulator includes three phases. Phase one takes the PSL specification data file as input, and changes the data file into proper format for the processing of phase two. He must use the command as follows.

.ph1 data-file-name

After phase one finishes, he may key in the next command for phase two as follows.

.ph2

Phase two of CPMS is like the processing of compiler or interpreter. Mainly, it checks the syntax of the input data file specified by PSL, and it can find out all errors except the syntax error of transition action statements. Through the display of the interactive terminal, the designer can find out syntax errors if any. Repeatedly, he modifies the data file and executes phase 1 and phase 2 until no error can be detected. And then, the designer may use the command as follows to execute phase 3.

.ph3

At phase 3 of CPMS, simulation is to be done. First, the system will ask the designer about "How many termination times need to be done ? ", and "Which modules need to be

simulated ? ".

The designer must answer the questions prompted. The termination times of the first question means the times of the system simulated to go into deadlock condition or the system simulated to match the condition of termiantion specified by the designer in the termination declaration. The second question means that the designer can indicate which module will be simulated, each module has a unique name and is specified at the first line of module declaration.

At phase 3, when syntax errors take place at the statements of transition actions, the simulator will terminate immediately. In case of errors, we must modify the PNDM and simulate it again from phase 1.

At phase 3, the simulator will provide the information about the sequence of transition firing, the conditions of deadlock, and condition of termination and forcing, and the conflict between transition firings. Some properties may not be able to be found out in our simulator. However, the designer may observe the sequences of transition firing, and find out some special conditions happening, e.g. looping and completeness.

5. The Alternating Bits Protocol

In this chapter, we will take the alternating bit protocol as an example to describe the sequence of using the CPMS simulator.

5.1 The alternating bits protocol

The alternating bits protocol is provided for a simple but reliable message transfer service over an unreliable transmission medium. It uses an one-bit sequence number for each message sent. The sequence number is complemented on each new message sent. When the receiver receives messages correctly, it transmits the acknowledgement to the sender. If the acknowledgement is not received by the sender within a timeout period, the same message will be retransmitted. The protocol guarantees correctly sequenced delivery of messages even if the medium lose messages or acknowldegements.

To accomplish these functions, the sender and the receiver stations must maintain local sequence number counters respectively. The sender uses a counter to maintain the sequence number which is the sequence number of the last transmitted message, and if the acknowledgement is received, the counter will be complemented and used as the sequnce number for the next message. The receiver also uses a counter to maintain the sequence number of the next message expected to received. This mechanism can remove the duplicate messages.

5.2 Modeling the alternating bits Protocol by PNDM

First, we add new transitions ML and AL to represent the message and acknowledgement transmission loss as shown in Fig 5.1.

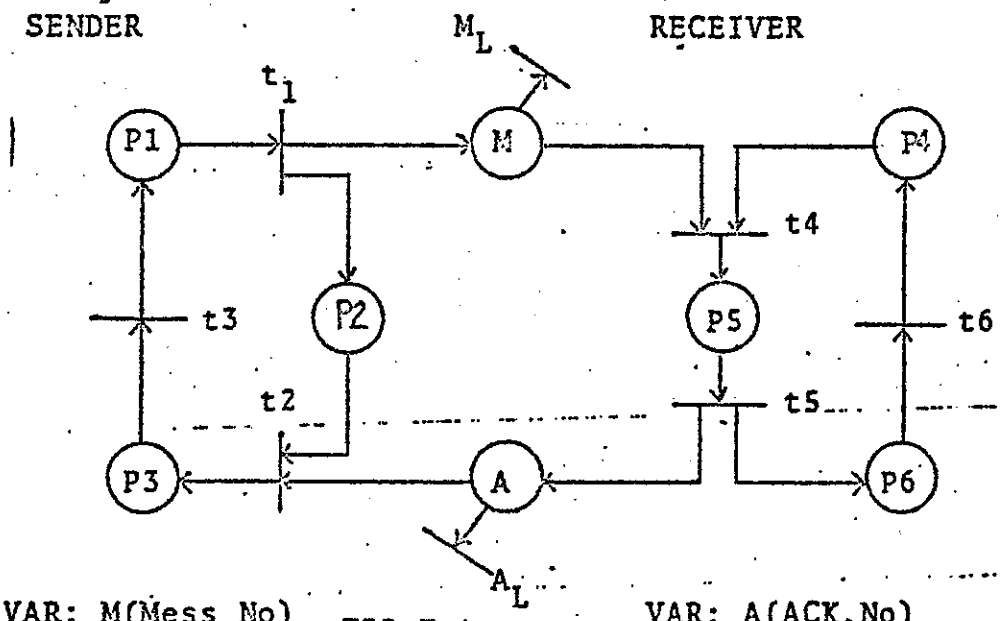


FIG 5.1

Here, we use counter M and counter A to represent the local counter of sender and receiver, respectively. In addition, we use the counter R to represent the number of retransmission times.

We use the transition actions to describe the transition hierarchy to extend the protocol in Fig 5.1. Transition t1, t2, t4, and t5 have more deep meaning than before. The action of transition t2, t4, and t5 are replaced by transition action statements as follows.

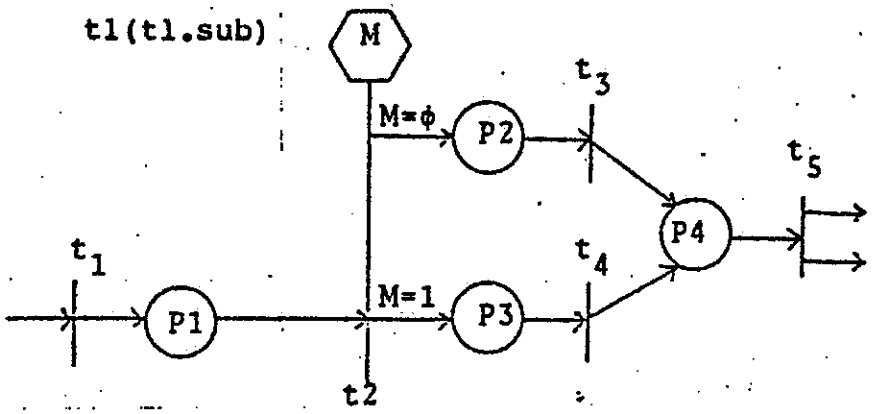
```
t4      receive 1 mess
        exit
```

```
        if (mess¬no == %1) ax<¬1
        if (mess¬no == %0) ax<¬0
        if (ax == a) ++a

t2      receive 2 ack
        exit
        ++m
        r <¬ 0
        <¬ timer t01

t5      send 2 ack
```

From the logic viewpoint, we are only concerned about the sequence of transition firing, and the data is not important in this example. So we only consider one bit data which is the sequence number of messages. At transition t4, when the sequence number of the message received is correct, the local counter A at receiver will be complemented by statement ++a. At transition t2, when the sequence number of acknowledgement received is correct, the local counter M at sender will be complemented by statement ++m and the retransmission times counter will be reset by statement r <¬ 0. At transition t5, the receiver sends the acknowledgement to the sender.

The action of transition t1 may be extended to submodule t1.sub by employing the concept of transition hierarchy

as follows.




```
t1.t2   when(t1.p1*m==0:t1.p2|t1.p1*m==1:t1.p3)

t1.t3   mess_no=%0
        send 1 mess_no

t1.t4   mess_no=%1
        send 1 mess_no

t1.t5   ++r
        if (r>5) stop
        timer t01 <- 0
```

The actions of transition t1 send the message with sequence number determined by the value of local counter M. When the number of retransmission times is over five, the system will be stopped, and the timer is started by statement timer t01 <- 0 at transition t1.t5.

5.3 Converting Representation from PNDM to PSL

After the protocol has been specified by PNDM, we must transform it into the form of PSL. We must first declare the places, transitions, and variables used, and then specify the termination conditions and forcing functions. Finally we describe the structures and actions of each transition.

Appendix C lists the alternating bits protocol described with PSL.

5.4 Using the CPMS simulator

To use CPMS, we edit the specification of the alternating bit protocol into a data file called "ABP". And then, execute the following commands sequentially.

```
.ph1 ABP
.ph2
.ph3
```

Through these three commands, the simulation can be run, and the results will be displayed at the interactive terminal.

Appendix D is a result of CPMS simulator phase 2. Appendix DR is a result of CPMS simulator phase 3.

Appendix A : General Form of Protocol Specification Language

```
MODULE module name (modb)

PLACE DECLARATION (decp)
 placename ; ... ... ;placename ;
END CF PLACE DECLARATION (edclp)

TRANSITION DECLARATION (dclt)
 transition name;... ...;transition name;
END OF TRANSITION DECLARATION (edclt)

TYPE DECLARATION (dcly)
 type name = type;
   .          .
   .          .
   .          .
END OF TYPE DECLARATION (edcly)

VARIABLE DECLARATION (dclv)
 STRUCTURE (structure)
  variable name : type name; ... ;
 COUNTER (ctr)
  counter variable name : (modulus value); ... ;
 TIMER COUNTER (tmctr)
  timer variabl name : (modulus value); ... ;
END OF VARIABLE DECLARATION (edclv)
```

```
MODULE INITIALIZATION (modi)
 place name : token number ;
    .          .
    .          .
    .          .
VAR variable name <= assign value ; ... ;
END OF MODULE INITIALIZATION (emodi)

MODULE TERMINATION (modt)
 TERMINATION termination name (term)
  place name   : token number ;
  .             .
  .             .
  .             .
 END OF TERMINATION termination name (eterm)
 .
 .
 .
END OF MODULE TERMINATION (emodt)

MODULE FORCING (modf)
 FORCING forcing name (forcing)
  placename : tokens number ;
    .          .
    .          .
 TIMEOUT timer counter name (timeout); ... ;
```

```
FORCE placename : token numeber ; (force)
   .           .
   .           .
END OF FORCING forcing name (eforcing)
.
.
.
END OF MODULE FORCING (emodf)

TRANSITION DESCRIPTION (dest)
 <TRANSITION DESCRIPTION BODY>
END OF TRANSITION DESCRIPTION (edest)

END OF MODULE module name (emodb)
```

```
TRANSITION DESCRIPTION (dest)

TRANSITION transition name (tran)
 RECEIVE ARC (rarc)
   placename (arc number) ;
        .             .
        .             .
  END OF RECEIVE ARC (erarc)
```

TRANSMIT ARC (tarc)

placename (arc number) ;

. .

. .

. .

END OF TRANSMIT ARC (etarc)

TRANSITION ACTION (actt)

1.when statement

2.counter initiate, increase.

3.timer start, destory.

4.if statement.

4.1.counter comparing

4.2.non-counter comparing

5.send, receive statements

6.exit, stop statements

7.assignment statement

END OF TRANSITION ACTION (eactt)

[<MODULE>]

END OF TRANSITION transition name (etran)

.

.

.

END OF TRANSITION DESCRIPTION (edest)

Appendix B : BNF of protocol specification language

<module>::=<module title><module body><module tail>

<module title>::=modb <module name>

<module tail>::=emodb <module name>

<module name>::=<name>

<module body>::=<place declaration><transition declaration>
[<type declaration>][<variable declaration>]
[<module initialization>][<module termination>]
[<module forcing>][<transition description>]

<place declaration>::=<place decl title><place decl body>
<place decl tail>

<place decl title>::=dclp

<place decl tail>::=edclp

<place decl body>::=<place name>;|<place decl body><place name>;

<transition declaration>::=<tran decl title><tran decl body>
<tran decl tail>

<tran decl title>::=dclt

<tran decl tail>::=edclt

<tran decl body>::=<transition name>;|<tran decl body>
<transition name>;

<transition name>::=<name>

<place name>::=<name>

<type declaration>::=<type decl title><type decl body>
<type decl tail>

<type decl title>::=dcly

<type decl tail>::=edcly

<type decl body>::=<type definition>;|<type decl body>
<type definition>;

<type definition>::=<type defi name>=<type>

<type defi name>::=<name>

<type>::=<simple type>|<block type>

<simple type>::=B1(<index integer>)|B8(<index integer>)
|B16(<index integer>)

<index integer>::=<integer>|<null>

<null>::=

<record type>::={<record section>}

<record section>::=<field name>=<type1>;|<record section>
<field name>=<type1>

<field name>::=<name>

<type1>::=<simple type>|<type defi name>

<variable declaration>::=<var decl title><var decl body>
<var decl tail>

<var decl title>::=dclv

<var decl tail>::=edclv

<var decl body>::=<structure var decl><counter var decl>
<timer var decl>
<structure var decl>::=structure <str var decl body>
<str var decl body>::=<variable name> : <type1>;
|<str var decl body><variable name> : <type1>;
<counter var decl>::=ctr <counter var decl body>
<counter var decl body>::=<counter name>(<counter modulus value>);
|<counter var decl body><counter name>
(<counter modulus value>);
<counter name>::=<name>
<variable name>::=<name>
<counter modulus value>::=<integer>

<timer var decl>::=tmctr <timer var decl body>
<timer var decl body>::=<timer name>(<timer modulus value>);
|<timer var decl body><timer name>(<timer modulus value>);
<timer name>::=<name>
<timer modulus value>::=<integer>

<module initialization>::=<module init title><module init body>
<module init tail>
<module init title>::=modi
<module init tail>::=emodi
<module init body>::=<place name>:<token number>;|<module init
body><place name>:<token number>;

<token number>::=<integer>

<module termination>::=<module termi title><module termi body>
<module term tail>

<module termi title>::=modt

<module termi tail>::=emodt

<module termi body>::=<module termi>|<module termi body>
<module termi>

<module termi>::=<module tbody title><module tbody>
<module tbody tail>

<module tbody title>::=term <term name>

<module tbody tail>::=eterm <term name>

<module tbody>::=<place name>:<token number>;|<module tbody>
<place name>:<token number>;

<term name>::=<name>

<module forcing>::=<module forcing title><module forcing body>
<module forcing tail>

<module forcing title>::=modf

<module forcing tail>::=emodf

<module forcing body>::=<module forcing>|<module forcing body>
<module forcing>

<module forcing>::=<module fbody title><module fbody>
<module fbody tail>

<module fbody title>::=forcing <forcing name>

<module fbody tail>::=eforcing <forcing name>

<module fbody>::=<fbody1>;force <fbody2>

<fbody2>::=<place name>:<token number>;|<fbody2><place name>
:<token number>;

<fbody1>::=<fbody2>|<fbody1>; timeout <timer name>;

<forcing name>::=<name>

<transition description>::=<tran desc title><tran desc body>
<tran desc tail>

<tran desc title>::=dest

<tran desc tail>::=edest

<tran desc body>::=<tr body title><tr body><tr body tail>|
<tr body title><tr body><tr body tail><tran desc body>

<tr body title>::=tran

<tr body tail>::=etran

<tr body>::=<tr rec arc><tr trs arc><tr action>|
<tr rec arc><tr trs arc><module>

<tr rec arc>::=rarc ; <tr recarc body> ; erarc ;

<tr recarc body>::=<place name>(<arc number>);|<tr recarc body>
<place name>(<arc number>);

<tr trs arc>::=tarc ; <tr trsarc body> ; etarc ;

<tr trsarc body>::=<place name>(<arc number>);
|<tr trsarc body><place name>(<arc number>);

<tr action>::=actt ; <tr action body> ; eactt ;

<tr action body>::=<when statement>|<statements group>

<when statement>::=when(<when action>)

<when action>::=<place name>*<counter comparing>:<place name>|
<when action>'|'<place name>*<conuter comparing>:<place name>

<statements group>::=<statement>|<statement group><statement>

<statement>::=<counter statement>|<timer statement>|
<if statement>|<terminate statement>|<transfer statement>
|<assignment statement>

<counter statement>::=<counter assigned statement>
|<counter increased statement>

<counter assigned statement>::=<counter name> "<-" <assigned value>

<assigned value>::=<integer>

<counter increased statement>::= ++ <counter name>

<timer statement>::=<timer start statement>|
<timer destory statement>

<timer start statement>::=timer <timer name> "<- 0"

<timer destory statement>::=stop | exit

<transfer statement>::=<send statement>|<receive statement>

<send statement>::=send <port no><message name>

<receive statement>::=receive <port no><message name>

<port no>::=1|2

<message name>::=<variable name>

<if statement>::=if (<comparing>) <statement> else <statemont>

<comparing>::=<counter comparing>|<noncounter comparing>

<counter comparing>::=<counter name><comparator><counter name>
|<counter name><comparator><integer>

<noncounter comparing>::=<variable name>"=="<variable name>
|<variable name>"=="<value of variable>

<comparator>::= < | > | == | >= | <=

<assignment statement>::=<bits assign statement>|<byes
assign statement>|<words assign statement>

<bytes assign statement>::=<structure variable name>="
<bytes string>

<bytes string>::=<digit>|<char>|<byte string><digit>|<byte
string><char>

<bits assign statement>::=<structure variable name>=%
<bits string>

<bits string>::=0|1|<bits string>0|<bit string>1

<words assign statement>::=<structure variable name>=
<words string>

<words string>::=<integer>|<word string> " " <integer>

<structure variable name>::=<variable name>|<structure variable
name>.<field name>

<value of variable>::=%<bits string>|"<bytes string>|<word string>

<name>::=<name><char>|<char>

<integer>::=<digit>|<integer><digit>

<char>::=a|b|c|...|z

<digit>::=0|1|2|...|9

Appendix C : Alternating Bits Protocol Described by PSL

```
modb test                    * module name

dclp                         * place declaration
p1;p2;p3;p4;p5;m;a;p6;
edclp

dclt                         * transition declaration
t1;t2;t3;t4;t5;t6;m1;a1;
edclt

dclv                         * variable declaration
structure                    * structure variable declaration
mess¬no:bl(1)
ack¬no:bl(1)
ctr a(2);m(2);r(5);          * counter declaration
 mx(2);ax(2);
tmctr t01(8);                * timer declaration
edclv

modi                         * module initialization
p3:1;p6:1;
var m<¬0
a<¬0
```

```
r<-0
emodi

modt                    * module termination condition
term test1
p1:1;
p4:1
otherm test1
emodt

modf                    * module forcing
forcing test1
timeout t01
force p1:1
p2:0
p3:0
oforcing test1
forcing test2
p1:1
p2:2
timeout t01
force p1:2
p2:2
p3:2
p4:4
```

```
eforcing test2
emodf

dest                        * transition description
tran t6                     * transition t6 description
rarc                        * receive arc
p6
erarc
tarc                        * trasmit arc
p4
etarc
etran t6

tran t3                     * transition t3 description
rarc
p3
erarc
tarc
p1
etarc
etran t3

tran m1                     * transition m1 description
rarc
m
```

```
erarc
tarc
etarc
etran m1

tran a1                    * transition a1 description
rarc
a
erarc
tarc
etarc
etran a1

tran t4                    * transition t4 description
rarc
p4;m;
erarc
tarc
p5
etarc
actt                       * transition action
receieve 1 mess;
exit;
if (mess-no==%1) ax<-1;
if (mess-no ==%0) ax<-0;
```

```
if (ax==a) ++a;
eactt
etran t4

tran t2                    * transition t2 description
rarc
a;p2;
erarc
tarc
p3
etarc
actt
receieve  2 ack;
exit;
++m
r <- 0
<-timer t01
eactt
etran t2
tran t5                    * transition t5 description
rarc
p5
erarc
tarc
a;p6;
```

```
etarc
actt
send 2 ack
eactt
etran t5

tran t1                    * transition t1 description
rarc
p1
erarc

tarc
p2;m;
etarc

modb t1.sub                * transition hierarchy
dclp                       * describing transition t1
t1.p1;t1.p2;t1.p3;t1.p4;
edclp

dclt
t1.t1;t1.t2;t1.t3;t1.t4;t1.t5;
edclt

dest
```

```
tran t1.t1                    * transition t1.t1 description
rarc
p1
erarc
tarc
t1.p1
etarc
etran t1.t1

tran t1.t5                    * transition t1.t5 description
rarc
t1.p1
etarc
tarc
p2;m;
etarc
actt
;
if (r>=5) stop;
timer t01 <= 0;
eactt
etran t1.t5

tran t1.t2                    * transition t1.t2 description
rarc
```

```
tl.pl
erarc
tarc
tl.p2
tl.p3
etarc
actt
when(tl.pl*m==0:tl.p2 | tl.pl*m==1:tl.p3)
eactt
etran tl.t2

tran tl.t3              * transition tl.t3 description
rarc
tl.p2
erarc
tarc
tl.p4
etarc
actt
mess-no=%0
send  1 mess-no;
eactt
etran tl.t3

tran tl.t4              * transition tl.t4 description
```

```
rarc
t1.p3
erarc
tarc
t1.p4
etarc
actt
mess-no=%1
send 1 mess-no
eactt
etran t1.t4

edest
emodb t1.sub                    * end of module transition t1
etran t1

edest
emodb test
```

Appendix D : Result of CPMS simulator phase 2 with ABP

```
read one data line : modb test
module name : test

read one data line : dclp
place declaration

read one data line : p1
placename : p1

read one data line : p2
placename : p2

read one data line : p3
placename : p3

read one data line : p4
placename : p4

read one data line : p5
placename : p5

read one data line : m
placename : m

read one data line : a
placename : a

read one data line : p6
placename : p6

read one data line : edclp
end of place declaration

read one data line : dclt
transition declaration

read one data line : t1
trans. name : t1

read one data line : t2
trans. name : t2

read one data line : t3
trans. name : t3

read one data line : t4
trans. name : t4
```

```
read one data line : t5
trans. name : t5

read one data line : t6
trans. name : t6

read one data line : m1
trans. name : m1

read one data line : a1
trans. name : a1

read one data line : edclt
end of transition declaration

read one data line : dclv
variable declaration

read one data line : structure
structure variable :

read one data line : mess_no:bl(1)
structure : mess_no:bl(1)

read one data line : ack_no:bl(1)
structure : ack_no:bl(1)

read one data line : ctr a(2)
counter :     a(2)

read one data line : m(2)
counter : m(2)

read one data line : r(5)
counter : r(5)

read one data line :  mx(2)
counter :  mx(2)

read one data line : ax(2)
counter : ax(2)

read one data line : tmctr t01(8)
timercounter :       t01(8)

read one data line : edclv
end of variable declaration

read one data line : modi
module initialization
```

```
read one data line : p1:1
¬¬place name : token number¬¬
p1:1

read one data line : p4:1
p4:1

read one data line : var m<¬Ø
¬¬variable name <¬¬ assign value¬¬
    m<¬Ø

read one data line : a<¬Ø
a<¬Ø

read one data line : r<¬Ø
r<¬Ø

read one data line : emodi
end of module initialization

read one data line : modt
module termination

read one data line : emodt
end of module termination

read one data line : modf
module forcing

read one data line : forcing test1
forcing with name : test1

read one data line : timeout tØ1
timeout :          tØ1

read one data line : force p1:1
force :       p1:1

read one data line : p2:Ø
force : p2:Ø

read one data line : p3:Ø
force : p3:Ø

read one data line : eforcing test1
end of forcing with name : test1

read one data line : forcing test2
forcing with name : test2
```

```
read one data line : p1:1
p1:1

read one data line : p2:2
p2:2

read one data line : timeout t01
timeout :          t01

read one data line : force p1:2
force :       p1:2

read one data line : p2:2
force : p2:2

read one data line : p3:2
force : p3:2

read one data line : p4:4
force : p4:4

read one data line : eforcing test2
end of forcing with name : test2

read one data line : emodf
end of module forcing

read one data line : dest
transition description

read one data line : tran t6
transition with name : t6

read one data line : rarc
receive arc

read one data line : p6
rec. arc name : p6

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : p4
transmit arc name : p4

read one data line : etarc
```

```
end of transmit arc

read one data line : etran t6
end of transition with name : t6

read one data line : tran t3
transition with name : t3

read one data line : rarc
receive arc

read one data line : p3
rec. arc name : p3

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : p1
transmit arc name : p1

read one data line : etarc
end of transmit arc

read one data line : etran t3
end of transition with name : t3

read one data line : tran m1
transition with name : m1

read one data line : rarc
receive arc

read one data line : m
rec. arc name : m

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : etarc
end of transmit arc

read one data line : etran m1
end of transition with name : m1
```

```
read one data line : tran a1
transition with name : a1

read one data line : rarc
receive arc

read one data line : a
rec. arc name : a

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : etarc
end of transmit arc

read one data line : etran a1
end of transition with name : a1

read one data line : tran t4
transition with name : t4

read one data line : rarc
receive arc

read one data line : p4
rec. arc name : p4

read one data line : m
rec. arc name : m

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : p5
transmit arc name : p5

read one data line : etarc
end of transmit arc

read one data line : actt
transition action

read one data line : receieve 1 mess
receieve 1 mess
```

```
read one data line : exit
exit

read one data line : if (mess-no==%1) ax<-1
if (mess-no==%1) ax<-1

read one data line : if (mess-no ==%0) ax<-0
if (mess-no ==%0) ax<-0

read one data line : if (ax==a) ++a
if (ax==a) ++a

read one data line : eactt
end of transition action

read one data line : etran t4
end of transition with name : t4

read one data line : tran t2
transition with name : t2

read one data line : rarc
receive arc

read one data line : a
rec. arc name : a

read one data line : p2
rec. arc name : p2

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : p3
transmit arc name : p3

read one data line : etarc
end of transmit arc

read one data line : actt
transition action

read one data line : receieve  2 ack
receieve  2 ack

read one data line : exit
```

```
exit

read one data line : ++m
++m

read one data line : r <₹ 0
r <₹ 0

read one data line : <₹ timer t01
<₹ timer t01

read one data line : eactt
end of transition action

read one data line : etran t2
end of transition with name : t2

read one data line : tran t5
transition with name : t5

read one data line : rarc
receive arc

read one data line : p5
rec. arc name : p5

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : a
transmit arc name : a

read one data line : p6
transmit arc name : p6

read one data line : etarc
end of transmit arc

read one data line : actt
transition action

read one data line : send 2 ack
send 2 ack

read one data line : eactt
end of transition action
```

```
read one data line : etran t5
end of transition with name : t5

read one data line : tran t1
transition with name : t1

read one data line : rarc
receive arc

read one data line : p1
rec. arc name : p1

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : p2
transmit arc name : p2

read one data line : m
transmit arc name : m

read one data line : etarc
end of transmit arc

read one data line : modb t1.sub
module name : t1.sub

read one data line : dclp
place declaration

read one data line : t1.p1
placename : t1.p1

read one data line : t1.p2
placename : t1.p2

read one data line : t1.p3
placename : t1.p3

read one data line : t1.p4
placename : t1.p4

read one data line : edclp
end of place declaration

read one data line : dclt
transition declaration
```

```
read one data line : t1.t1
trans. name : t1.t1

read one data line : t1.t2
trans. name : t1.t2

read one data line : t1.t3
trans. name : t1.t3

read one data line : t1.t4
trans. name : t1.t4

read one data line : t1.t5
trans. name : t1.t5

read one data line : edclt
end of transition declaration

read one data line : dest
transition description

read one data line : tran t1.t1
transition with name : t1.t1

read one data line : rarc
receive arc

read one data line : p1
rec. arc name : p1

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : t1.p1
transmit arc name : t1.p1

read one data line : etarc
end of transmit arc

read one data line : etran t1.t1
end of transition with name : t1.t1

read one data line : tran t1.t5
transition with name : t1.t5

read one data line : rarc
```

```
receive arc

read one data line : t1.p4
rec. arc name : t1.p4

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : p2
transmit arc name : p2

read one data line : m
transmit arc name : m

read one data line : etarc
end of transmit arc

read one data line : actt
transition action

read one data line : ++r
++r

read one data line : if (r>=5) stop
if (r>=5) stop

read one data line : timer t01 <- 0
timer t01 <- 0

read one data line : eactt
end of transition action

read one data line : etran t1.t5
end of transition with name : t1.t5

read one data line : tran t1.t2
transition with name : t1.t2

read one data line : rarc
receive arc

read one data line : t1.p1
rec. arc name : t1.p1

read one data line : erarc
end of receive arc
```

```
read one data line : tarc
transmit arc

read one data line : t1.p2
transmit arc name : t1.p2

read one data line : t1.p3
transmit arc name : t1.p3

read one data line : etarc
end of transmit arc

read one data line : actt
transition action

read one data line : when(t1.p1*m==0:t1.p2 | t1.p1*m==1:t1.p3)
when(t1.p1*m==0:t1.p2 | t1.p1*m==1:t1.p3)

read one data line : eactt
end of transition action

read one data line : etran t1.t2
end of transition with name : t1.t2

read one data line : tran t1.t3
transition with name : t1.t3

read one data line : rarc
receive arc

read one data line : t1.p2
rec. arc name : t1.p2

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : t1.p4
transmit arc name : t1.p4

read one data line : etarc
end of transmit arc

read one data line : actt
transition action

read one data line : mess-no=%0
mess-no=%0
```

```
read one data line : send  1 mess
send  1 mess

read one data line : eactt
end of transition action

read one data line : etran t1.t3
end of transition with name : t1.t3

read one data line : tran t1.t4
transition with name : t1.t4

read one data line : rarc
receive arc

read one data line : t1.p3
rec. arc name : t1.p3

read one data line : erarc
end of receive arc

read one data line : tarc
transmit arc

read one data line : t1.p4
transmit arc name : t1.p4

read one data line : etarc
end of transmit arc

read one data line : actt
transition action

read one data line : mess#no=%1
mess#no=%1

read one data line : send 1 mess
send 1 mess

read one data line : eactt
end of transition action

read one data line : etran t1.t4
end of transition with name : t1.t4

read one data line : eqest
end of transition description

read one data line : emodb t1.sub
```

```
successfully

read one data line : etran t1
end of transition with name : t1

read one data line : edest
end of transition description

read one data line : emodb test
successfully
```

Appendix DR : Result of CPMS simulator phase 3 with ABP

```
How many terminations times do you want ?
This module--test need simulate (y|n) ?
This module--t1.sub need simulate (y|n) ?

place p1 hold 1 token
place p2 hold 0 token
place p3 hold 0 token
place p4 hold 1 token
place p5 hold 0 token
place m hold 0 token
place a hold 0 token
place p6 hold 0 token
place t1.p1 hold 0 token
place t1.p2 hold 0 token
place t1.p3 hold 0 token
place t1.p4 hold 0 token

transition t2 is selected to simulate.
transition t3 is selected to simulate.
transition t4 is selected to simulate.
transition t5 is selected to simulate.
transition t6 is selected to simulate.
transition m1 is selected to simulate.
transition a1 is selected to simulate.
transition t1.t1 is selected to simulate.
transition t1.t2 is selected to simulate.
transition t1.t3 is selected to simulate.
transition t1.t4 is selected to simulate.
transition t1.t5 is selected to simulate.

*****************************
*****************************
    simulation start
*****************************
*****************************
transition t1.t1 is enabled


transition t1.t1 firing
absorb 1 tokens from place p1
send 1 token to place : t1.p1
transition t1.t2 is enabled


transition t1.t2 firing
```

```
absorb 1 tokens from place t1.p1
send 1 token to place : t1.p2
send 1 token to place : t1.p3
action statement : when(t1.p1*m==0:t1.p2 | t1.p1*m==1:t1.p3)
when statement action
transition firing reversing
send 1 token to place : t1.p1
absorb 1 tokens from place t1.p2
absorb 1 tokens from place t1.p3
absorb 1 tokens from place : t1.p1
send 1 tokens to place : t1.p2
transition t1.t3 is enabled


transition t1.t3 firing
absorb 1 tokens from place t1.p2
send 1 token to place : t1.p4
action statement : mess=no=%0
bits assignment
action statement : send  1 mess
send message
transition t1.t5 is enabled


transition t1.t5 firing
absorb 1 tokens from place t1.p4
send 1 token to place : p2
send 1 token to place : m
action statement : ++r
counter increasing
action statement : if (r>=5) stop
if condition is false
action statement : timer t01 <= 0
start timer counter
transition t4 is enabled
transition m1 is enabled
```

```
timer counter t01 has value 7
timer decreasing


transition t4 firing
absorb 1 tokens from place p4
absorb 1 tokens from place m
send 1 token to place : p5
--------------------------------
action statement : receieve 1 mess
receive message
--------------------------------
--------------------------------
action statement : if (mess->no==%1) ax<-1
if condition is false
--------------------------------
--------------------------------
action statement : if (mess->no ==%0) ax<-0
if condition is true
counter assignment
--------------------------------
--------------------------------
action statement : if (ax==a) ++a
if condition is true
counter increasing
--------------------------------
transition t5 is enabled

transition m1 conflict with previous transition firing


transition t5 firing
absorb 1 tokens from place p5
send 1 token to place : a
send 1 token to place : p6
--------------------------------
action statement : send 2 ack
send message
--------------------------------
transition t2 is enabled
transition t6 is enabled
transition a1 is enabled

timer counter t01 has value 6
timer decreasing
```

```
transition t2 firing
absorb 1 tokens from place a
absorb 1 tokens from place p2
send 1 token to place : p3
**********************************
action statement : receieve  2 ack
receive message
**********************************
**********************************
action statement : ++m
counter increasing
**********************************
**********************************
action statement : r <= 0
counter assignment
**********************************
**********************************
action statement : <= timer t01
destory timer counter
**********************************
transition t3 is enabled
transition t6 is enabled

transition a1 conflict with previous transition firing


transition t6 firing
absorb 1 tokens from place p6
send 1 token to place : p4
transition t3 is enabled


transition t3 firing
absorb 1 tokens from place p3
send 1 token to place : p1
transition t1.t1 is enabled


transition t1.t1 firing
absorb 1 tokens from place p1
send 1 token to place : t1.p1
transition t1.t2 is enabled


transition t1.t2 firing
```

```
absorb 1 tokens from place t1.p1
send 1 token to place : t1.p2
send 1 token to place : t1.p3
action statement : when(t1.p1*m==0:t1.p2 | t1.p1*m==1:t1.p3)
when statement action
transition firing reversing
send 1 token to place : t1.p1
absorb 1 tokens from place t1.p2
absorb 1 tokens from place t1.p3
absorb 1 tokens from place : t1.p1
send 1 tokens to place : t1.p3
transition t1.t4 is enabled


transition t1.t4 firing
absorb 1 tokens from place t1.p3
send 1 token to place : t1.p4
action statement : mess-no=%1
bits assignment
action statement : send 1 mess
send message
transition t1.t5 is enabled


transition t1.t5 firing
absorb 1 tokens from place t1.p4
send 1 token to place : p2
send 1 token to place : m
action statement : ++r
counter increasing
action statement : if (r>=5) stop
if condition is false
action statement : timer t01 <- 0
start timer counter
transition t4 is enabled
transition m1 is enabled
```

```
timer counter t01 has value 7
timer decreasing

timer counter t01 has value 6
timer decreasing

timer counter t01 has value 5
timer decreasing

timer counter t01 has value 4
timer decreasing

timer counter t01 has value 3
timer decreasing



transition m1 firing
absorb 1 tokens from place m

transition t4 conflict with previous transition firing

timer counter t01 has value 2
timer decreasing

timer counter t01 has value 1
timer decreasing

timer counter t01 has value 0
timer decreasing

timer timeout

forcing action
place p2 hold 1 token
place p4 hold 1 token
=========================
after action
place p1 hold 1 token
place p4 hold 1 token
=========================
transition t1.t1 is enabled



transition t1.t1 firing
absorb 1 tokens from place p1
send 1 token to place : t1.p1
transition t1.t2 is enabled
```

```
transition t1.t2 firing
absorb 1 tokens from place t1.p1
send 1 token to place : t1.p2
send 1 token to place : t1.p3
--------------------------------
action statement : when(t1.p1*m==0:t1.p2 | t1.p1*m==1:t1.p3)
when statement action
transition firing reversing
send 1 token to place : t1.p1
absorb 1 tokens from place t1.p2
absorb 1 tokens from place t1.p3
absorb 1 tokens from place : t1.p1
send 1 tokens to place : t1.p3
--------------------------------
transition t1.t4 is enabled


transition t1.t4 firing
absorb 1 tokens from place t1.p3
send 1 token to place : t1.p4
--------------------------------
action statement : mess-no=%1
bits assignment
--------------------------------
--------------------------------
action statement : send 1 mess
send message
--------------------------------
```