TECHNISCHE HOGESCHOOL DELFT

Afd. der Elektrotechniek

| | | |
|---|---|---|
| Aard | : | rapport |
| Omvang | : | 48 p. |
| Datum | : | Augustus 1983 |
| Lab | : | Automatische Verkeerssystemen |
| Nummer | : | no. 45 |
| Auteur | : | dr.ir. Gerard J. Holzmann |
| Titel | : | Automated protocol verification on Pandora: Four examples |

Delft University of Technology
Department of Electrical Engineering
AVS Laboratory
Mekelweg 4
Delft, 2600 GA
The Netherlands

**CONTENTS**

*[The text of this report was stored on an 8" floppy disk from my old TRS80 Model II computer from 1983. To produce this pdf, the text was imported into MS-Word and the original layout reproduced as best as possible. The page-numbers above were updated to match this version. GJH, 28 July 2007.]*

# Automated protocol verification on PANDORA
Four examples

Gerard J. Holzmann

Delft University of Technology
The Netherlands

ABSTRACT

The working of the Pandora protocol development system is illustrated with a discussion of four communication protocols of varying size and complexity. Three of the four examples have a rich tradition serving as guinea pigs for new protocol validation methods. The fourth example illustrates the use of Pandora for the analysis of process interactions in larger systems.

The examples chosen are:
(1) the NPL alternating bit protocol,
(2) the Arpanet three-way handshake protocol,
(3) CCITT recommendation X-21, and
(4) the P2 protocol, a draft specification of a new CCITT protocol for the implementation of a higher level message handling service.

Keywords: automated verification, communication protocol, formal modeling, protocol specification, protocol compilation, protocol validation, Pandora.

Delft, August 1983

## 1. Introduction

The Pandora system [Ho '83, Ho&Be '83] was designed to provide programmers with a new set of tools specifically for the development of communication protocols.

To assess the adequacy of these tools we have experimented with the analysis of a small set of protocols whose working and validity is richly documented in the literature, supplemented with one larger protocol that was still in an early design phase at the time of analysis.

The emphasis in this report will be placed on the guidance provided by the Pandora system in the protocol design process by the protocol compiler and by the protocol analyzer, that can help a user improve a design step by step until compiler, analyzer and user all three are satisfied.

## 2. The NPL alternating bit protocol [Ba '69]

The alternating bit protocol must be one of the simplest, best documented, and most thoroughly verified protocol designs around. Some recent tests were documented in an excellent comparative study of four different automated protocol verification systems performed at USC/ISI [Su&Sm '82].

Despite the thoroughness with which the protocol has been verified on numerous systems with as many different methods, its popularity has not diminished: one out of every three papers submitted to a recent conference on protocol specification [Ru&We '83] contained a discussion of the alternating bit protocol.

Since verifying the correctness of the alternating bit protocol is turning into a required exercise for new protocol validation methods we will described here, what may well be called, "yet another verification of the alternating bit protocol."

### 2.1 Specification

The protocol validation process in the Pandora system is based on an algebraic analysis of the protocol interaction grammar [Ho '82ab]. An explicit restriction of this analysis method is that value transfers, being no part of the grammar, are normally not modeled. Unfortunately, in the alternating bit protocol the value transfer of the one bit sequence number must be considered an essential part of the protocol to be analyzed. So, we seem to have a problem.

In this case, the effective range of the sequence number is rather small, so we can simply include the sequence number into the message names we use to force the analyzer to take it into account. In section 3 we will show another way to work our way around the value transfer restrictions in Pandora, but first things first.

The alternating bit protocol can be modeled in three processes: the receiver, the sender, and the channel between sender and receiver.

We will first discuss a meta specification for the sender process:

```
        proc sender
                do
                :: SEQ1; SEQ0
                od
        end;
```

The sender process executes a do-loop, alternately calling the reference tasks SEQ1 and SEQ0. In the first task the sequence number used in sends is 1, in the second task it is 0.

The specification of reference task SEQ1 can be written as:

```
        ref sender: SEQ1
                channel!msg1;
                do
                :: channel?ack1 -> break
                :: default -> channel!msg1
                od
        end;
```

The sender will transmit a message 'msg1' to the channel process and then wait for a response. If the response is message 'ack1' (representing an acknowledgement with sequence number 1) the loop is broken and the task completed. Any other message that arrives will be caught in the 'default' clause, which will cause the original message 'msg1' to be retransmitted and the loop to be reentered from the top.

Reference task SEQ0 is derived from SEQ1 by replacing all 1's by 0's. The specification of the receiver process is very similar to that of the sender. We use two more reference tasks: EXPECT1 and EXPECT0.

```
        proc receiver
                do
                :: EXPECT1; EXPECT0
                od
        end;
```

with the following definition for task EXPECT0:

```
        ref receiver: EXPECT0
                do
                :: channel?msg0 -> channel!ack0; break
                :: default -> channel!ack1
                od
        end;
```

Task EXPECT1 is again derived from EXPECT0 by replacing 1 for 0 and the two 0's for two 1's.

That leaves the specification of the channel process. The precise behavior that we ascribe to the channel will, of course, determine the results of our analysis. For an ideal channel we cannot expect to find more than a confirmation that the protocol will faithfully transfer messages from sender to receiver, without disturbances. We can, however, equally well model a channel that randomly duplicates, deletes, or disturbs messages. In the following example specification of the channel process we have modeled a behavior in which one specific message (ack0) is randomly distorted by the channel, while the others are allowed to complete their journeys between sender and receiver undisturbed.

```
proc channel
        do
        :: sender?msg0 -> receiver!msg0
        :: sender?msg1 -> receiver!msg1
        :: receiver?ack1 -> sender!ack1
        :: receiver?ack0 ->
                if              /* random choice  */
                :: sender!ack0   /* error-free     */
                :: sender!xxx    /* simulate error */
                fi
        od
end channel.
```

**2.2 Analysis**

The analysis of the above specification will be done in two steps: first an extensive compile-time analysis for completeness and consistency, then the full run-time analysis of the interaction grammar. The first step is performed by the protocol compiler (proco), the second step by the protocol analyzer (pan).

In verbose mode, the listing generated by the protocol compiler is as follows:

```
PROTOCOL OVERVIEW:
 3      processes:
                 1   channel
                 2   sender
                 3   receiver
 4      reference tasks:
                 1   sender:SEQ1
                 2   sender:SEQ0
                 3   receiver:EXPECT0
                 4   receiver:EXPECT1

MESSAGES EXCHANGED:
 1      sender->channel:msg1
 2      channel->sender:ack1
 3      sender->channel:msg0
 4      channel->sender:ack0
 5      channel->receiver:msg0
 6      receiver->channel:ack0
 7      receiver->channel:ack1
 8      channel->receiver:msg1
 9      channel->sender:xxx

 0      timeouts, and 4 defaults

HINTS AND WARNINGS:
Warning: channel->sender:xxx is sent but not received
```

The compile time analysis is performed in 5.2 sec of CPU time (on a PDP 11/24). The compiler warns us that the sender process cannot recognize the mutilated message 'xxx', which indeed is what we intended. The message will be caught by the sender in a 'default' clause when it arrives.

The next step is the run-time analysis performed by 'pan'. The analyzer uses a conversion of the abstract specification into protocol formulas, prepared by 'proco' during the compile time analysis. The algebraic analysis takes 9.6 sec of CPU time. Pan's verdict on the protocol is:

```
pan: performing analysis for an empty initial string
pan: 3 processes and 3 message queues
pan: no deadlocks found, no residuals, 1 loop detected
```

The first line reminds us that we did not specify any initial contents of the message queues for the initial state in which the analysis is to begin. The second line indicates the size of the protocol being analyzed. Each process can receive messages via up to nine independent queues. In the above example, though, we have used just one queue per process.

The third line tells us what we were after: the protocol cannot deadlock, when all processes have returned to their initial states all message queues are empty (no residuals), but, there is also one potentially unproductive loop in which the protocol may hang.

Reassured that the alternating bit protocol indeed contains no deadlocks we can print the one suspect looping execution sequence. It looks as follows:

```
1:
MESSAGE     sender channel receiver

msg1            ----->
msg1                   ----->
ack1                   <-----
ack1            <-----
msg0            ----->
msg0                   ----->
ack0                   <-----
****
xxx             <-----
msg0            ----->
msg0                   ----->
ack0                   <-----
****
ack0            <-----
====

End listing
```

Four stars are used to indicate the start end the end of the partial execution sequence that can be repeated indefinitely. Each time the body of this loop is executed the message msg0 is retransmitted, and the acknowledgement ack0 transformed into a message xxx by the channel. It confirms that the alternating bit protocol may hang in an unproductive loop when the channel process will maliciously and consistently disturb all ack0 messages it sees.

The analyzer will also produce a file with a listing of "dead code" in the protocol specification. In this case we could easily have foreseen that some of the default clauses cannot be executed. The only default clause that does fulfill its role is in task SEQ0.

We can now proceed with the analysis by experimenting with different channel behaviors. We can thus find out quickly what the performance of the alternating bit protocol is for channels that can duplicate or delete messages.

Clearly, to resist a channel that may lose messages we have to include timeouts in the specification. We can, for instance, experiment with the following alternative specification of the alternating bit protocol.

```
        proc sender
                do
                :: SEQ1; SEQ0
                od
        end;

        proc receiver
                do
                :: EXPECT1; EXPECT0
                od
        end;

        ref sender: SEQ1
                receiver!msg1;
                do
                :: receiver?ack1 -> break
                :: default -> receiver!msg1
                :: timeout -> receiver!msg1
                od
        end;

        ref sender: SEQ0
                receiver!msg0;
                do
                :: receiver?ack0 -> break
                :: default -> receiver!msg0
                od
        end;

        ref receiver: EXPECT0
                do
                :: sender?msg0 -> sender!ack0; break
                :: default -> sender!ack1
                od
        end;

        ref receiver: EXPECT1
                do
                :: sender?msg1 -> sender!ack1; break
                :: default -> sender!ack0
                od
        end.
```

This time, we have assumed an error-free channel. We could have altered the specification of the channel process to model such a channel, but since it no longer has any function in the model it is simpler to omit it completely. We soon discover, however, that the timeouts can again create duplicate messages. These duplicates turn out to be the real problem with the alternating bit protocol. The duplicate messages never disappear and tend to add an ever-increasing overhead to the communication.

The analyzer insists that also this version of the protocol is free from deadlocks, but this time there are residuals and several suspect loops. Four characteristic [Ho '83] loop sequences are listed by the analyzer. One is the trivial sequence:

```
        MESSAGE       sender receiver
        msg1            -------->
        ***
        timeout         +
        msg1!           -------->>
        ***
        ===
```

The double arrow indicates a message sent that is not immediately received, but appended to a message queue. The plus indicates the reception of a message without an immediately preceding send action.

It should be noted here that the value of the timeout count is immaterial to the analysis. Only the possibility of timeouts is taken in the account, not their probability [Ho '82].

One of the three other sequences generated is:

```
        MESSAGE       sender receiver
        msg1!           -------->>
        ***
        msg1?                   +
        timeout         +
        ack1!          <<--------
        msg1            -------->
        ack1?           +
        ack1!          <<--------
        msg0            -------->
        ack1?           +
        ack0!          <<--------
        msg0            -------->
        ack0?           +
        msg1!           -------->>
        ack0!          <<--------
        ***
        ......
```

Each time through the loop produces one ack0 message more than is consumed, and the same can clearly happen with other messages. Whether or not one should call the protocol "correct" probably depends on the faith one has in the behavior of the channel and in the accuracy of the timeout counts chosen.

## 3. The Arpanet three-way handshake [Sch '81]

### 3.1 Specification

With an analysis of the three-way handshake protocol we can illustrate in a little more detail how we can analyze protocols in which the effect of value-transfers is important.

The purpose of the three-way handshake protocol is to reach agreement between sender and receiver on the values of certain protocol parameters, in this case the initial values of message sequence numbers. Apart from general issues such as freedom of deadlock, we would, for instance, like to verify that the data transfer phase of the protocol cannot be entered unless proper agreement on the sequence numbers has been reached, knowing that the channel between the two parties can arbitrarily delay, duplicate, delete or distort messages.

Let us first, quite unrealistically, assume that we are dealing with an error free channel and see what a minimal specification of the three-way handshake would look like. We will name the two communicating parties process 'this' and process 'that'.

```
proc this
     if
     :: that?synM_ -> that!synackNM; that?ackMN
     :: that!synN_ ->
             if
             :: that?synackMN -> that!ackNM
             :: that?synM_ -> that!ackNM; that?ackMN
             fi
     fi;

     do
     ::  that!data -> that?data
     od
end this;

proc that
     if
     :: this?synN_ -> this!synackMN; this?ackNM
     :: this!synM_ ->
             if
             :: this?synackNM -> this!ackMN
             :: this?synN_ -> this!ackMN; this?ackNM
             fi
     fi;

     do
     :: this!data -> this?data
     od
end that.
```

Negotiations start when one side transmits the synchronization message 'syn' with a proposal for the initial sequence number. In the above specification this is symbolized by attaching the symbol N or M to the message name. Since the 'syn' message is the first message transmitted the acknowledgement field is empty, which is indicated by an underscore.

The reception of a 'syn' message is acknowledged by the remote side with the sending of a 'synack' message carrying two values: (1) an acknowledgement of the sequence number received via the 'syn' message, and (2) the remote side's proposal for a sequence number to be used on the return channel. The three-way handshake is completed if the sender confirms the reception of the synack by sending an 'ack' message, carrying both values picked for the sequence numbers.

Running this simple model through the analyzer yields 3 suspect execution loops. The data transfer phase can be entered via three different routes, and once data transfer is entered the protocol hangs:

```
pan: performing analysis for an empty initial string
pan: 2 processes and 2 message queues
pan: no deadlocks found, no residuals, 3 loops detected

1:
MESSAGE       this   that

synN_           ----->
synackMN      <-----
ackNM           ----->
****
data!           ----->>
data          <-----
data?                +
****
====

2:
MESSAGE      this   that

synN_!          ----->>
synM_         <-----
synN_?               +
ackNM!          ----->>
ackMN         <-----
ackNM?               +
****
data!           ----->>
data          <-----
data?                +
****
====

3:
MESSAGE      this   that

synM_         <-----
synackNM        ----->
```

```
        ackMN           <-----
        ****
        data!            ----->>
        data            <-----
        data?                 +
        ****
        ====

        End listing
```

Since this is the behavior we intended, we can now proceed by modifying the specification for more realistic assumptions about the channel behavior.

It is convenient at this point to make explicit a number of of relevant protocol states, similar to [Sch '81]. We will therefore first replace the above description with the equivalent, though somewhat longer:

```
   proc this
   closed:  if
            :: that?synM_ -> that!synackNM; goto synrec
            :: that!synN_ -> goto synsent
            fi;

   synsent: if
            :: that?synackMN -> that!ackNM; goto estab
            :: that?synM_ -> that!ackNM; goto synrec
            fi;

   synrec:  that?ackMN -> goto estab;

   estab:   do
            :: that!data -> that?data
            od
   end this;
```

And similarly for process 'that'.

Though it seems that we haven't changed much in the specification as such, we have gained a few 'warnings' from the protocol compiler 'proco' for the use of the labels 'closed' in the main body of the processes 'this' and 'that' without matching gotos:

```
        Warning: this/_main:closed is an unused label
        Warning: that/_main:closed is an unused label
```

More important, though, is that the code for the four relevant protocol phases can be located more easily in this listing.

We can now extend the specification with a process modeling the communication link between the two processes:

```
   proc channel
        if
        :: this?synN_ -> that!synN_
        :: this?synackNM -> that!synackNM
        ... etc.
        fi
   end channel
```

This channel process can serve as a temporary holder of all messages passed between processes this and that, which will now address all their messages to channel instead of directly to each other. It is easy to fake a malicious channel simply by tampering with the specification of the channel process. For instance, we can duplicate messages randomly by saying:

```
        :: this?synN_ ->
                if
                :: that!synN_              /* normal response */
                :: that!synN_; that!synN_  /* duplicated mesg */
                fi
        ... etc.
```

Or even worse, we may insert spurious messages into the stream by adding non-conditional options like:

```
        :: that!synX_        /* send without first receiving */
        ... etc.
```

In both cases we should be prepared to receive out-of-context messages in processes 'this' and 'that'. The last specification we have given will therefore have to be extended with the proper response for each such message, without, however, opening up new spurious paths into the data transfer phase (state 'estab').

The channel process can thus be used to model specific erroneous environments that the protocol must be able to resist.

The analysis task can be simplified considerably, though, if we note that the channel is merely a 'modifier' of the behavior of the party a 3way handshake agent is communicating with. This modified behavior may or may not be in accordance with the official protocol as given above, and our problem is to make the 3way handshake agent resist any attempts to establish data transfers other than via the prescribed procedure.

So we can easily do away with the channel process and with one of the two 3way handshake agents. In the analysis we can restrict ourselves to just one official 3way handshake specification, and one malicious communication partner that generates valid and invalid messages at random:

```
   proc malicious
        do
        :: this!synM_
        :: this!synackMN
        ... etc.
        od
   end malicious
```

A agent implementing the 3way handshake that simply ignores any out-of-context message, may then look as follows:

```
   proc this
   closed:  do
            :: that?synM_ -> that!synackNM; goto synrec
            :: that!synN_ -> goto synsent
            :: default -> skip
            od;

   synsent: do
            :: that?synackMN -> that!ackNM; goto estab
            :: that?synM_ -> that!ackNM; goto synrec
            :: default -> skip
            od;

   synrec:  do
            :: that?ackMN -> goto estab
            :: default -> skip
            od;

   estab:   do
            :: that!data -> that?data
            :: default -> skip
            od
   end this;
```

In this example, the protocol agent waits in each state until a proper message comes in, ignoring all others no matter how many illegal messages are received.

Clearly, in the long run, such an agent will always reach the data transfer phase when communicating with processes that produce an endless stream of random messages. To avoid this, the real 3way handshake protocol generates a 'reset' message for most out-of-context messages received and makes a forced return to state 'closed'. To check for out-of-context messages it will also verify the validity of the acknowledgement and sequence numbers tagged to the messages.

Note that the 'synN_' message carries a sequence number N that cannot always be verified by the receiver since it normally is the first such number submitted.

The 'synackMN' message acknowledges the initiators sequence number N and submits a new sequence number M. Clearly, the initiator of the protocol session can validate the first number, but usually not the second.

The numbers tagged to the 'ackMN' message, finally, can always both be validated.

> Although we do not really have variables we can still perform the analysis required here by using just two symbols for each parameter: a valid one (named N or M) and an invalid one (which we will name X). Our 'malicious' process can generate a random sequence of messages, again randomly tagging valid or invalid parameters to them. All we have to do is check that no sequence of invalid messages can disturb the call setup.

The new version we will analyze is based on [Sch '81]. It is the complete protocol specification, but we have omitted one option in state 'synsent', for brevity: in the real 3way handshake protocol each agent can make a shortcut to state 'estab' upon the reception of a new 'syn' message if the agent's sequence number was acknowledged before (e.g. if this 'syn' is a duplicate of an earlier one). We can verify below that omitting this option does not introduce new errors.

Here is the complete specification we will analyze, with process 'that' posing as the 'malicious' side.

```
   proc this
   closed:
        do
        :: that!synN_ -> goto synsent
        :: that?synM_ -> that!synackNM; goto synrec
        :: that?resetM_ -> skip
        :: that?resetX_ -> skip     /* invalid seq nr */
        :: default -> that!resetN_
        od;

   synsent:
        do
        :: that?synackMN -> that!ackNM; goto estab
        :: that?synM_ -> that!ackNM; goto synrec
        :: that?synX_ -> that!ackNM; goto synrec
                                /* shortcut to estab omitted */
        :: that?synackMX -> that!resetN_   /* invalid ack nr */
        :: that?synackXX -> that!resetN_   /* invalid ack nr */
        :: that?ackMX -> that!resetN_      /* invalid ack nr */
        :: that?ackXX -> that!resetN_      /* invalid ack nr */
        :: that?resetM_ -> goto close
        :: default -> skip
        od;

   synrec:
        do
        :: that?ackMN -> goto estab

        :: that?resetM_ -> goto closed
        :: that?resetX_ -> goto closed     /* invalid seq nr */

        :: that?ackMX -> that!resetN_      /* invalid ack nr */
        :: that?ackXX -> that!resetN_      /* invalid ack nr */
        :: that?ackXN -> that!ackNM        /* invalid seq nr */

        :: that?synackMX -> that!resetN_   /* invalid ack nr */
        :: that?synackXX -> that!resetN_   /* invalid ack nr */
        :: that?synackXN -> that!ackNM     /* invalid seq nr */

        :: that?synX_ -> that!ackNM   /* dup, invalid seq nr */

        :: that?synM_ -> skip
        :: that?synackMN -> skip
        :: that?data -> that!resetN_
        od;

   estab:
        that!data;
        if
        :: that?data
        :: default -> that!resetN_
        fi
   end this;

   ref that:template
        if
        :: this!synM_
        :: this!synX_
        :: this!ackMN
        :: this!ackMX
        :: this!ackXN
        :: this!ackXX
        :: this!synackMN
        :: this!synackMX
        :: this!synackXN
        :: this!synackXX
        :: this!resetM_
        :: this!resetX_
        :: this!data
        fi
   end template;

   proc that
        do
        :: this?data -> this!data; break
        :: this?resetN_ -> skip
        :: default -> template
        od
   end that.
```

This time, some out-of-context messages are ignored (e.g. a reset message in state 'closed'), some generate a reset message but do cause a state change (anything other than a reset in state 'synsent'), and some do both.

The message exchange stops if either side reaches the data transfer phase. Malicious process 'that' will randomly pick a response to any message that comes in other than 'data' or 'reset'. We've programmed this with the aid of a reference task 'template'.

A remaining problem with the modeling of value transfers is illustrated in the treatment of the response to the message 'synX_' in state 'synsent'. The protocol does not allow us to to distinguish between valid and invalid sequence numbers here, so if the invalid 'synX_' comes in (perhaps an old duplicate from a previous incarnation of the protocol) we have to treat it as a valid one. For lack of a better solution, this has been modeled by mapping the invalid 'X' onto a valid symbol 'M'.

## 3.2 Analysis

Running the above specification through the analyzer yields the following disconcerting message from 'pan':

```
        pan: performing analysis for "empty" initial string
        pan: 2 processes and 2 message queues
        pan: 23 deadlocks found, residuals, 6 loops detected
```

So, what's wrong?

Let us first examine the following 8 deadlock sequences from among the 23 claimed to be uncovered by pan.

```
        1:
        MESSAGE      this   that

        synN_           ----->
        synM_           <-----
        ackNM           ----->
        synM_           <-----
        ====

        4:
        MESSAGE      this   that

        synN_           ----->
        synM_           <-----
        ackNM           ----->
        synackMN        <-----
        ====

        8:
        MESSAGE      this   that

        synN_           ----->
        synX_           <-----
        ackNM           ----->
        synM_           <-----
        ====

        11:
        MESSAGE      this   that

        synN_           ----->
        synX_           <-----
        ackNM           ----->
        synackMN        <-----
        ====

        15:
        MESSAGE      this   that

        synN_           ----->
        ackMN           <-----
        ====

        17:
        MESSAGE      this   that

        synN_           ----->
        ackXN           <-----
        ====
        20:
        MESSAGE      this   that

        synN_           ----->
        synackXN        <-----
        ====

        22:
        MESSAGE      this   that

        synN_           ----->
        data            <-----
        ====
```

From among these eight, the first five sequences are most interesting since they do not involve any (detectable) out-of-sequence message.

Checking the sequences against the protocol reveals that a deadlock state is reached when process 'that' sends a message that is ignored by the protocol agent process 'this'. True, the real 3way handshake comes with an implicit timeout mechanism that retransmits any outstanding message if no response is received within a reasonable period of time, but this won't help process 'that' at all. (Process 'that' may end up retransmitting the ignored message *ad infinitum*.)

The remaining 15 deadlock sequences reported by 'pan' all involve properly detected out-of-sequence messages that trigger a reset message from 'this', but do not cause a change of state. Note that process 'that' does not respond to 'reset' messages, which causes the message exchange to block at that point. (We could, of course, equally well have process 'that' echo the reset message to send process 'this' back to state 'closed'.)

Now let us look at the six 'suspect loops' detected by pan, using the Pandora command 'showloops'.

```
1:
MESSAGE      this   that

synN_           ----->
synM_          <-----
ackNM           ----->
****
synX_          <-----
ackNM           ----->
****
====

2:
MESSAGE      this   that

synN_           ----->
synM_          <-----
ackNM           ----->
****
ackXN          <-----
```

```
        ackNM           ----->
        ****
        ====

        3:
        MESSAGE      this   that

        synN_           ----->
        synM_          <-----
        ackNM           ----->
        ****
        synackXN       <-----
        ackNM           ----->
        ****
        ====

        4:
        MESSAGE      this   that

        synN_           ----->
        synX_          <-----
        ackNM           ----->
        ****
        synX_          <-----
        ackNM           ----->
        ****
        ====

        5:
        MESSAGE      this   that

        synN_           ----->
        synX_          <-----
        ackNM           ----->
        ****
        ackXN          <-----
        ackNM           ----->
        ****
        ====

        6:
        MESSAGE      this   that

        synN_           ----->
        synX_          <-----
        ackNM           ----->
        ****
        synackXN       <-----
        ackNM           ----->
        ****
        ====

        End listing
```

Fortunately, all six turn out to be 'starvation loops' for cases in which process 'that' consistently picks the same message that does not cause a change of state in the receiver.

The loops are a byproduct of the particular model of the 'malicious' protocol agent we've chosen here; they reveal little about the protocol as such.

One final point remains to be considered. We still have to show what we set out to prove: that the data transfer phase cannot be entered without proper agreement about the protocol parameters. We can ask the system for a listing of characteristic sequences that correspond to 'normal executions' (no detected errors). (Command: 'showall'.) Just three sequences come up that do no include any reset message. They are:

```
1:
MESSAGE      this   that

synN_          ----->
synM_         <-----
ackNM          ----->
ackMN         <-----
data           ----->
data          <-----
====

4:
MESSAGE      this   that

synN_          ----->
synX_         <-----
ackNM          ----->
ackMN         <-----
data           ----->
data          <-----
====

17:
MESSAGE      this   that

synN_          ----->
synackMN      <-----
ackNM          ----->
data!          ----->>
data          <-----
data?               +
data!        <<-----
====
```

The second listing looks all wrong. It is caused by the peculiar mapping of the invalid sequence number 'X' onto the valid one 'M', commented on earlier in section 3.1.  Apart from this impurity we see just two ways for the protocol to reach the data transfer phase: by the mimicking of either an entire 'active' or 'passive' opening sequence in complete accordance with the 3way-handshake protocol by malicious process 'that'.

We have noted that a shortcut to state 'estab' was omitted from the specification we have analyzed, claiming that it would cause no new errors in the protocol. In fact, omitting the shortcut really 'improves' the protocol by preventing errors from occurring. Schwabe

[Sch '81] detected a liveness error in the protocol with the shortcut was omitted from our version. With the shortcut, the following sequence of events is possible:

```
        MESSAGE      this   that

        synM_           X---     ( message from 'that' is lost )
        synN_          ------>   ( message comes through       )
        ackMN        <------     ( 'synsent': 'ack' ignored    )
        synX_        <-----o     ( an old 'syn' duplicate      )
        ackNX        ------->    ( 'this' responds to synX_    )
        data         <-----o     ( 'this' accepts invalid data  )
        ...
        reset        <------     ( reset clears the session     )
```

Process 'this' can take the shortcut because the 'synX_' message is not the first 'valid' message it receives. Note that without the shortcut, process 'this' is forced to wait for the right 'ack' message before accepting the 'data'.

## **4. The X.21 Protocol** [We '78, Ra '80, Ho '81]

By virtue of its simplicity, the X.21 protocol has proven itself to be another helpfull exercise for assessing the usefulness of protocol testers.

An analysis of X.21 with the Pandora system quickly teaches us that a fair amount of inventiveness may still be required in the construction of a useful model of the protocol under test, and that some care may be required in the interpretation of the analysis results. The main problem we must face is that X.21 assumes low level, unbuffered communications (the exchange of raw signals on electrical circuitry). The Pandora system, however, was designed mainly for the analysis of higher level, buffered communications. We must therefore start with some assumptions about the implementations of X.21 that our analysis will apply to. One possible implementation studied by [We '78] was described as:

> ... a modular design in which each processor contained an input decoder, whose function was to monitor the signals on the input circuits, assemble characters and messages, and indicate to a decision making unit when a complete input signal or message had been received. [We '78, p. 68]

We will make one extension to these assumptions: we will also assume that the 'input decoder' will buffer incoming messages.

The buffer assumption rules out one specific type of errors: we will not catch errors that may occur if signals are sent that may 'overtake' and thereby nullify previously sent signals. On the other hand, some of the 'errors' in a buffered version of X.21 that Pandora will report will most certainly have to be disqualified for unbuffered implementations. In particular this will be true for detected 'buffer locks' (that is: unspecified receptions that may block call progress indefinitely).

With these provisos we ar ready to undertake the analysis of X.21.

**4.1 Specification**

In X.21 both sides of the protocol may initiate clear requests at virtually any time during call set-up. In an attempt to simplify our initial analysis efforts, we will ignore the jumps to the call clearing phase for a while. First we will try to find out if errors may occur during the normal call set-up and clear-down procedures defined by X.21.

The following discussion is loosely based on the analysis reported in [Ho '81], this time using the more recent and more powerful implementation of the protocol analyzer 'pan' embedded in the Pandora system.

As in [Ho '81], we use the state diagrams from [We '78]. The names of all messages exchanged have been replaced by letters for conciseness. The program-like description that follows is again most naturally made in a state oriented fashion. Since the maximum number of gotos and labels that we are allowed to use by Pandora's protocol compiler is restricted, many labels that merely serve as reference points, but are not used otherwise, have been commented out to hide them from the compiler.

We start with a specification of the 'dte' side of the protocol.

```
        /*** Specification of the 'dte' side: ***/

        proc dte
        state01:
                do
        /*      :: dce!b -> dce!a       -- jump disabled */
                :: dce!i -> /* state14 */
                            if
                            :: dce?m -> goto state19
                            :: dce!a
                            fi
                :: dce?m -> goto state18
                :: dce?u -> goto state08
                :: dce!d -> break
                od;

        /* state02: */
                if
                :: dce?v
                :: dce?u -> goto state15
                :: dce?m -> goto state19
                fi;
        state03:
                dce!e;
        /* state04: */
                if
                :: dce?m -> goto state19
                :: dce!c
                fi;
        /* state05: */
                if
                :: dce?m -> goto state19
                :: dce?r
                fi;
        /* state6A: */
                if
                :: dce?m -> goto state19
                :: dce?q
                fi;
        state07:
                if
                :: dce?m -> goto state19
                :: dce?r
                fi;
        /* state6B: */
                if
                :: dce?q -> goto state07
                :: dce?q
                :: dce?m -> goto state19
                fi;
        /* state10: */
                if
                :: dce?m -> goto state19
                :: dce?r
                fi;
        state6C:
                if
                :: dce?m -> goto state19
                :: dce?l
                fi;
        /* state11: */
                if
                :: dce?m -> goto state19
                :: dce?n
                fi;
        /* state12: */
                if
                :: dce?m -> goto state19
                :: dce!b -> goto state16
                fi;
        state15:
                if
                :: dce?v -> goto state03
                :: dce?m -> goto state19
                fi;
        state08:
                if
                :: dce!c
                :: dce!d -> goto state15
                :: dce?m -> goto state19
                fi;
        /* state09: */
                if
                :: dce?m -> goto state19
                :: dce?q
                fi;
        /* state10B: */
                if
                :: dce?r -> goto state6C
```

```
                :: dce?m -> goto state19
                fi;
        state18:
                if
                :: dce?l -> goto state01
                :: dce?m -> goto state19
                fi;
        state16:
                dce?m; /* state17: */
                dce?l; /* state21: */
                dce!a; goto state01;
        state19:
                dce!b; /* state20: */
                dce?l; /* state21: */
                dce!a; goto state01
        end;
```

The 'dce' side is in many ways the mirror image of the above listing in which sends are replaced by receives and vice versa. Where an unconditional send action translates into a conditional receive action, however, we have included passive options for call clearing. Note that each side must always be prepared to receive call clear requests, even though they will not be generated randomly in this version.

```
        /*** Specification of the 'dce' side: ***/

        proc dce
        state01:
                do
                :: dte?b -> /* state21 */ dte?a
                :: dte?i -> /* state14 */
                                if
                                :: dte?b -> goto state16
                                :: dte?a
                                fi
        /*      :: dte!m -> goto state18   -- jump disabled */
                :: dte!u -> goto state08
                :: dte?d -> break
                od;

        /* state02: */
                if
                :: dte!v
                :: dte!u -> goto state15
                :: dte?b -> goto state16
                fi;
        state03:
                if
                :: dte?e
                :: dte?b -> goto state16
                fi;
        /* state04: */
                if
                :: dte?b -> goto state16
                :: dte?c
                fi;
        /* state05: */
        if
        :: dte?b -> goto state16
        :: dte!r
        fi;
/* state6A: */
        if
        :: dte?b -> goto state16
        :: dte!q
        fi;
state07:
        if
        :: dte?b -> goto state16
        :: dte!r
        fi;
/* state6B: */
        if
        :: dte!q -> goto state07
        :: dte!q
        :: dte?b -> goto state16
        fi;
/* state10: */
        if
        :: dte?b -> goto state16
        :: dte!r
        fi;
state6C:
        if
        :: dte?b -> goto state16
        :: dte!l
        fi;
/* state11: */
        if
        :: dte?b -> goto state16
        :: dte!n
        fi;
/* state12: */
        if
        :: dte?b -> goto state16
        :: dte!m -> goto state19
        fi;
state15:
        if
        :: dte!v -> goto state03
        :: dte?b -> goto state16
        fi;
state08:
        if
        :: dte?c
        :: dte?d -> goto state15
        :: dte?b -> goto state16
        fi;
/* state09: */
        if
        :: dte?b -> goto state16
        :: dte!q
        fi;
/* state10B: */
```

```
                if
                :: dte!r -> goto state6C
                :: dte?b -> goto state16
                fi;
        state18:
                if
                :: dte!l -> goto state01
                :: dte?b -> goto state16
                fi;
        state16:
                dte!m; /* state17: */
                dte!l; /* state21: */
                dte?a; goto state01;
        state19:
                dte?b; /* state20: */
                dte!l; /* state21: */
                dte?a; goto state01
        end.
```

**4.2 Analysis**

The analysis of this model takes just 100 seconds of CPU time on the 11/24. The result:

```
        pan: performing analysis for an empty initial string
        pan: 2 processes and 2 message queues
        pan: 3 deadlocks found, 1 residual, 2 loops detected
```

The loops detected by 'pan' are trivial:

```
        1:
        MESSAGE      dte    dce

        d!               ----->>
        u               <-----
        d?                    +
        v               <-----
        e               ----->
        c               ----->
        r               <-----
        q               <-----
        ****
        r               <-----
        q               <-----
        ****
        ====

        2:
        MESSAGE      dte    dce

        d               ----->
        v               <-----
        e               ----->
        c               ----->
        r               <-----
        q               <-----
        ****
```

```
r               <-----
q               <-----
****
====

End listing
```

We may have expected to find a complaint from 'pan' about the 'loop' via state 14 back to state 1. But, 'pan' correctly classifies it as a normal execution: it returns both sides to the initial state. The only problem with this execution is that the 'dte' side may traverse the loop more often than the 'dce' side. The effect of this is to be found in the 'residuals' report. 'Pan' reports a residual string consisting of the two messages 'I' and 'A'. This 'residual' is the potential left-over of a protocol execution in the message queues.

But on to more interesting matters. The following deadlock sequences are reported:

```
1:
MESSAGE      dte    dce

i!               ----->>
u!              <<-----
====

2:
MESSAGE      dte    dce

d!               ----->>
u               <-----
d?                   +
v               <-----
e               ----->
c               ----->
r               <-----
q               <-----
r               <-----
q               <-----
r               <-----
q!              <<-----
====

3:
MESSAGE      dte    dce

d               ----->
v               <-----
e               ----->
c               ----->
r               <-----
q               <-----
r               <-----
q               <-----
r               <-----
q!              <<-----
====
        End listing
```

It is interesting to compare these listings with the errors described in previous studies of X.21. In the important study [We '78] three types of error were located. One type of error concerned call clearing, which we have decided to ignore for the moment.

The main error uncovered in [We '78], and the first of the three types discussed there, is illustrated in listing number '1' above: a collision due to a transition by the 'dte' to a 'not ready' state. Our analysis also shows that this is the only error of this type. (The other examples that were claimed to be in this class by [We '78] are readily shown to be valid sequences, even by hand.)

Quite surprisingly in view of our restrictive assumptions about the protocol and its implementation, we seem not to have missed out on the other known errors. Our last two sequences above are errors belonging to the third class from [We '78], there attributed to the ambiguity (in their and our model) in the interpretation of, what we have named, message 'Q' in process 'dte': process 'dte' cannot determine whether an incoming 'Q' should lead to state 7 or to state 10. These errors are easily avoided when the ambiguity can be resolved.

***Call Clearing***

The full analysis, including all spontaneous call clear requests turns out to be more time-consuming, to say the least. This analysis, including the post-processing of all reported errors takes several hours of CPU time on the 11/24. As it turns out, the protocol as presented allows for a very large quantity of deep execution histories, in some cases of more than 128 consecutive message exchanges (the current search depth of 'pan'). Here is pan's final verdict after this lengthy job completes:

```
        pan: performing analysis for an empty initial string
        pan: 2 processes and 2 message queues
        pan: 145 deadlocks found, residuals, 5164 loops detected
        pan: 16 undiagnosed sequences (exceeds search depth of pan)
```

Just out of curiosity, we may look at the first deadlock sequence reported by 'pan':

```
        1:
        MESSAGE      dte    dce

        b!              ----->>
        m!             <<-----
        b?                   +
        a!              ----->>
        i!              ----->>
        m?            +
        b!              ----->>
        m!             <<-----
        ====
```

Though this certainly seems to be a legitimate error, there is, of course, no way to delve through all the listings of the hundreds of deadlocks and loops to classify them in any useful way. It may be that the use of message buffers, where every reordering of messages lined up creates a new error variant, backfires at last. It is, however, also safe to conclude that call clearing is not the strongest point of the X.21 protocol design.

## 5. The P2 protocol [Pri '83, Rey '83]

To conclude this report we will study the analysis of a draft protocol for the implementation of message handling services. The function of the P2 protocol is roughly to act as an intermediary between an entity called the 'user environment' and an entity called the 'message transfer layer'. In the analysis we can model the desired behavior of user environment and message transfer layer in process definitions. Together with these two 'phantom' processes the draft (prepared and first analyzed on Pandora by Hans den Reijer [Rey '83]) consists of fourteen different process definitions.

It is not our intention here to discuss the P2 protocol itself, or its basis, in any detail. As with the previous examples discussed in this report we will use the draft P2 protocol as an illustration of analysis and modeling techniques on the Pandora system. There are a number of problems to overcome in analyzing the draft P2 protocol that can be generalized readily to more standard problems in modeling larger communication protocols. In the following discussion we will therefore focus on these issues using the P2 protocol specification as our vehicle of thought.

Throughout the discussion that follows we refer to the full listing of the draft protocol, cast in the Pandora specification language, as given in appendix B.

### 5.1 Specification

#### 5.1.1 Modeling the environment

The two 'phantom' processes are used to trigger the actions of the twelve others by submitting requests and indications. The user environment, for instance, can submit three types of requests: 'sendrequest', 'receiverequest', and 'proberequest', each one relayed to a different internal process.

We could model this as follows, with a spontaneous option in a do-statement:

```
proc userenvironment
do
:: sendandnumer!sendrequest
:: receivingUAE!receiverequest
:: uprobe!proberequest
:: default -> skip
od
end;
```

The user environment will then send a random sequence of sendrequests, receiverequests and proberequests, each directed to another process, and it will ignore all responses.

Though it certainly is an intuitively clear way of handling this problem, it will not help us very much in the analysis. The protocol analyzer will immediately detect a wild execution loop that can flood the input buffers of the three processes 'sendandnumber', 'receivingUAE' and 'uprobe'. It will report the three loops that cause it (one for each unconditional option above) and halt. The draft protocol cannot help us yet in determining whether or not this type of behavior was intended or should be prohibited. The behavior of the user environment is outside the scope of the P2 draft. For the time being, though, we have to be more subtle in modeling the user environment.

Our next attempt can be to have the user environment wait for the proper response after each request has been submitted:

```
        proc userenvironment
        do
        :: sendandnumber!sendrequest ->
                        sendandnumber?sendconfirmation
        :: receivingUAE!receiverequest ->
                        receivingUAE?receiveconfirmation
        :: uprobe!uproberequest ->
                        uprobe?uprobeconfirmation
        :: default -> skip
        od
        end;
```

It's better now, but still not good enough. The default option is meant to absorb 'indication' messages from the message transfer layer, relayed via a number of internal processes such as the 'status' and the 'receivingUAE' process. It can be explicted in two options:

```
        :: status?statusindication -> skip
        :: receivingUAE?availableindication -> skip
```

Clearly, we will get errors when a 'statusindication' message arrives while the user environment is waiting for either one of the three 'confirmation' messages. So, we will have to resort to the following bulky process description:

```
    proc userenvironment
    do
    :: sendandnumber!sendrequest ->
           do
           :: status?statusindication
           :: receivingUAE?availableindication
           :: sendandnumber?sendconfirmation -> break
           od
    :: receivingUAE!receiverequest ->
           do
           :: status?statusindication
           :: receivingUAE?availableindication
           :: receivingUAE?receiveconfirmation -> break
           od
    :: uprobe!uproberequest ->
           do
           :: status?statusindication
           :: receivingUAE?availableindication
           :: uprobe?uprobeconfirmation -> break
           od
    :: status?statusindication
    :: receivingUAE?availableindication
    od
    end;
```

This seems to be a lot of overhead just to say that the user environment can generate three types of messages. The analysis will, however, now show that every request will trigger a confirmation; if not, the user environment will hang, and a deadlock will be reported.

> It would also have been possible that we were not at all interested in establishing this particular fact. In that case we could have omitted the entire user environment process and use the facility to prescribe an initial message queue contents for each analysis run. With the Pandora command 'init' we can then list a different combination of requests each time through the analysis, ignore the residuals in the (by the responses implicitly defined) message queue of the usr environment process, and see if anything else in the P2 protocol may go wrong. We will, however, not pursue this here.

The message transfer layer can submit two types of indications: 'deliverindication' and 'notifyindication', again both directed at a different internal process, which will pass it on toward the user environment. Again:

```
proc mtl
do
:: deliver!deliverindication
:: notify!notifyindication
:: default -> skip
od
end;
```

will not work. But, this time we are in more trouble. The message transfer layer will not get a 'confirmation' in return for an 'indication'. It is only restricted in the number of consecutive 'indication's that it may produce by another protocol that it uses to communicate to a remote party. That protocol, however, is not modeled here.

An easy, though not a very satisfactory, way out of this dilemma is to simulate the message transfer layer's behavior more restrictively, thus, ignoring responses:

```
ref sendeither
if
:: deliver!deliverindication
:: notify!notifyindication
fi
end;
```

```
        proc mtl
        if
        :: sendeither
        :: sendeither; sendeither
        fi
        end;
```

which will send either one or two consecutive messages, randomly selected from a set of two.

**5.1.2 Modeling the internal processes**

Seven of the remaining twelve processes in the P2 specification map into deceptively simple processes when modeled in the Pandora specification language. Here are two of those:

```
    proc notify
    do
    :: mtl?notifyindication -> storeID!signal15
    :: storeID?signal16     -> status!signal17
    od
    end;

    proc status
    do
    :: deliver?signal18 -> userenvironment!statusindication
    :: notify?signal17  -> userenvironment!statusindication
    :: submit?signal7   -> userenvironment!statusindication
    od
    end;
```

The other seven processes are named 'storeID', 'sendandnumber', 'deliver', 'autoforward', and 'receivingUAE' (see app. B). Each of these processes sits in a tight loop, waiting for any one from a small set of messages to arrive. Upon arrival the processes will perform some local computations which is not modeled here, and then respond by passing a new message to another process.

The protocol compiler can quickly establish whether or not the set of messages that each process is prepared to receive is complete, and if it is we can be sure that these processes cannot possibly cause any run time errors. Both of the above processes have just one wait state, and the messages received in that wait state form a complete set.

Strictly spoken, as far as the analyzer is concerned, these processes are utterly redundant. They are not only completely specified, their behavior is also completely predictable. We could save ourselves, and the analyzer, a lot of time and worries if we could simply bypass the redundant processes and generate their predictable response right away.

Such an 'optimization' should, of course, not been taken lightly. We have to examine very closely if there is any sound reason to bother at all with optimizing our specification if it doesn't change the behavior anyway. And secondly: if there is a sound reason for optimizing the specification we will have to be very careful that we are not optimizing away potential errors, or introducing spurious new ones.

## ***Why the 7 processes must go***

The first question then is: should we bother at all reducing the number of processes to please the analyzer. Superficially, there is no such reason. The analyzer can handle up to 16 concurrent processes, and will happily handle all the interactions specified.

But, if we do invoke the analyzer in this manner we soon find that it gives us more than ample time to consider why it takes so long before the results appear.

We will make two short side-trips to illustrate the difficulties encountered here.

### (1) Mailbox orderings

The analyzer strictly enforces the fifo discipline on all message queues, even if we are not at all interested in the order in which messages are appended to or retrieved from message queues. The seven processes we consider here have one and only one wait state in which they can receive any message that can come in. By virtue of their simplicity they do not use the fifo discipline. Nevertheless, the analyzer is forced to wade through ALL possible ways in which messages can line up for any single one of the seven suspects. To appreciate the complexity involved here, even for this relatively small number of processes, we have to consider the second point more closely.

### (2) Compexity Issues

To perform an automated analysis for a protocol of any size we have to walk the tightrope of combinatorial explosion, no matter what analysis method is used. Since the protocol analysis problem is provably 'undecidable' in general [e.g. Bra '83], we cannot hope for analysis methods that can tackle protocols of arbitrary complexity. (Note that the term 'complexity' is not to be confused with 'abstraction level'.) Most analysis methods work hard at trying to reduce the amount of work involved in an analysis so that the class of protocols that can effectively be analyzed is large enough to be of practical value. Especially the algebraic analysis method used in the Pandora system derives its major strength from the fact that it can achieve a near optimal amount of reductions by a careful process of eliminating equivalences [Ho '82].

The seven simple processes, however, can easily defeat the analyzer at this reduction game. The fifo mailbox ordering is the one item that the analyzer will never touch, and precisely this fifo mailbox ordering is what the seven processes ignore. Even for small process vocabularies (three or four messages per processes) the complexity of the analysis will predictably increase by several orders of magnitude. If the protocol without the 7 suspects could be analyzed in, say, ten minutes of CPU time, an increase of just three orders of magnitude would boost this to a full week of calculations. So, it seems that if there is even the slightest chance that we can safely omit the seven processes, no matter how deviously simple they look, we must certainly do so.

***Why the 7 processes can safely go***

Though it is not very difficult to remove the seven processes from the specification, we should, of course, first convince ourselves that we will not miss any errors by doing so. It is clear that the processes themselves cannot cause errors that we miss by deleting them: in this case the protocol compiler can establish their completeness (refer to the earlier discussion above). What will change is that some message will skip transfer points. This in turn may rule out a possibility that messages overtake one another on their way to a common destination.

The draft specification tells us that the seven processes can generate 15 different messages.

From the 15 messages 4 are directed to the user environment. Our final choice for the specification of the user environment has ruled out errors that are caused by one message overtaking another: all messages are accepted in all wait states, so there are no problems here.

From the remaining 11 messages 6 are directed at processes within the group of seven that stands to be eliminated. They can aggravate the problem of messages overtaking each other, but not alter the external behavior.

From the last 5 messages 2 are directed at process 'sendingUAE' and 3 are directed at process 'submit':

```
    sendingUAE!signal3:1
    sendingUAE!signal1
    submit!signal20
    submit!signal21
    submit!signal23
```

The two messages for 'sendingUAE' are the only two messages that this process can receive, so there could be an ordering problem here. It is not hard to verify that the problem is real, and indeed has been corrected in the draft specification by addressing each message at a different queue (the extension ':n' at the tail of a message name specifies message queue 'n' instead of the default queue '0'). The two wait states in process 'sendingUAE' and both take messages from a different queue:

```
    proc sendingUAE
         do
         ::   sendandnumber?signal1;
              if
              :: storeID!signal2 -> storeID?signal3:1
              :: skip
              fi;
              if
              :: submit!signal4
              :: submit!signal5
              fi
         od
    end;
```

So, no harm is done for process 'sendingUAE'. (Note also that by deleting process 'storeID' one of the two wait states will disappear.)

The same reasoning applies to the messages directed at process 'submit'. Again the problem of message overtaking existed, but was corrected before by using two message queues: one for the outer loop, and one for the inner loops (within the reference tasks, see appendix B).

**5.2 Analysis**

By recursively applying the same principle for eliminating processes, simultaneously checking to see if we can eliminate the sending of messages that are ignored by their recipients and by casting out any duplicates, we can effectively reduce the system of 14 processes to a set of only 3.

The only interesting problem that remains to be analyzed after all these manual reductions is expressed in the following specification:

```
proc user
do
:: handler!question ->
        if
        :: handler?answer
        :: handler?noanswer
        fi
od
end;

proc handler
do
:: user?question -> remote!request;
        if
        :: remote?reponse -> user!answer
        :: timeout -> user!noanswer
        fi
:: remote?reponse -> skip
:: default -> skip
od
end;

proc remote
do
:: handler?request -> handler!response
:: handler?request -> skip
od
end.
```

The remote process can either answer or ignore requests from the user, passed to it via the handler process. The handler process can timeout the remote process and decide that no answer is forthcoming. The problem here is in the precise coding of the handler process (cf. process 'submit' and process 'uprobe' in the P2 protocol, appendix B).

The handler process above cannot accept a second question before the first one has been processed completely.

To avoid this we could use two queues:

```
        proc handler
        do
        :: user?question -> remote!request;
                if
                :: remote?reponse:1 -> user!answer
                :: timeout:1 -> user!noanswer
                fi
        od
        end;
```

But this time we can erroneously timeout on the first question and interpret the delayed answer to be the answer for the second question, thus:

```
        MESG            user     handler     remote

        question          --------->
        request                    ------------>
        timeout:1                  +
        response:1                 <<----------
        noanswer          <---------
        question          --------->
        request                    ------------>
        response:1?                +
        answer            <---------
```

There are numerous variations on this theme, especially if we add explicit timing processes, as in the draft specification. An old timeout signal can then be mistaken for a new one, this time causing the untimely abortion of a waiting cycle.

There are no easy solutions to this problem, and indeed it seems to be the gravest problem with the draft P2 protocol.

The problem does not manifest itself in the form of an unspecified reception, a deadlock, or even an execution loop. It appears as a suspect residual of the first analysis run: some unprocessed responses and timing signals may stay behind to cause havoc in continued analyses. Though one is tempted to ignore the sometimes bulky residual listings, the example of the P2 protocol warns us against dismissing these results too lightheartedly.

## 6. Conclusions

Perhaps of more importance than the flaws that our analyses with the Pandora system has helped us reveal in the four protocols that we have discussed here is the approach to protocol analysis that it has forced us to adopt. There are probably few protocol designs that we will ever be faced with validating that will fit the straightjacket of an automated analyzer perfectly. We do, however, hope to have shown here that with the right mix of inventiveness and care we can go a long way in working with standardized tools.

The problem we faced in the analysis of the alternating bit protocol, and more seriously still with the 3way handshake protocol, was to model the effect of protocol parameters. By mapping only the relevant states of protocol parameters into message names we were able to defeat the restrictions the protocol analyzer tried to impose on us.

This treatment of protocol parameters is admittedly somewhat crude. It would be more elegant to use explicit variables in the protocol specification language, combined with a message format such as "synack(N,M)". But, alas, neither variables nor value transfers are within the scope of Pandora's specification language so we have to try and manage without them.

The analysis of X.21 has shown us a number of important things. It demonstrated how a seemingly complicated analysis can be broken down into manageable parts by reducing the scope of an initial attempt. But, it also illustrated both how hard protocol analysis can be, even with a tool as Pandora. We have already discussed some limitations of our analysis methods for low level protocols that are the result of discrepancies in the underlying model of communication (e.g. buffered or non-buffered).

Though the effort required from the user to enter a first abstracted model of the protocol, compile and debug it, and then perform a fast logic analysis is truly negligible, the effort required to go beyond this first step, to extend the scope and the value of the analysis is sometimes considerable. The user may be forced to write special purpose programs to select from, or to compress, an unmanageable amount of error sequences.

As in the case of the P2 draft protocol, it may also be difficult to 'tune' a specification to get the most out of an analysis without letting the analysis times get out of hand. Unfortunately, there are no general rules that will work in all cases.

An important conclusion concerning the Pandora system itself must be that the protocol compiler needs to be extended with an 'optimizer'. It should be considered in detail how much of the process of elimination and reduction that was so carefully motivated and then so painstakingly carried out manually for the P2 draft protocol, could also be automated. An effective and reliable optimization of this type will surely extend the range of protocols that can be analyzed with a minimum amount of human intervention.

A second conclusion concerning the Pandora system is that our early decision not to include value transfers in the specification language must be reevaluated. This problem, though, may well be much harder to solve correctly since the extension of the analyzer must be matched by an equally powerful extension of the underlying validation algebra. Again, a worthy topic for further research, but also one that is likely to be hard.

**Acknowledgement**

Hans de Reijer has done invaluable work preparing the specification of the P2 protocol for analysis on the Pandora system. I thank him for his patience with the early versions of the analysis and synthesis software. The design and construction of the Pandora system was made possible by a grant from the Netherlands PTT, under contract number 45155 OCA.

**REFERENCES**

[Ba '69] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson, A note on reliable full-duplex transmission over half-duplex links," *Comm. ACM*, Vol. 12, pp. 260-265, May 1969.

[Bra '83] D. Brand & P. Zafiropulo, On comunicating finite state machines, *Journal of the ACM*, Vol. 30, No.2, April 1983, pp. 323-342.

[Ho '81] G.J. Holzmann, An algebra for protocol validation. *Proc. First IFIP/INWG Workshop on Protocol Specification, Verification and Testing*, Ed. D. Rayner, NPL, England, May 1981.

[Ho '82a] G.J. Holzmann, A theory for protocol validation. *IEEE Trans. on Computers*, Vol. C-31, No. 8, August 1982, pp 730-738.

[Ho '82b] G.J. Holzmann, *Concise description of a protocol validation algebra.* Delft University of Technology, Report AVS Lab, No. 38, 1982.

[Ho '82c] G.J. Holzmann, *The Pandora system: reduction techniques*. Delft University of Technology, Report AVS Lab, No. 43, Dec. 1982.

[Ho '83] G.J. Holzmann, The Pandora system: an interactive system for the design of data communication protocols. To appear in *Computer Networks*, 1983.

[Ho&Be '83] G.J. Holzmann, and R.A. Beukers, The Pandora protocol development system. *Proc. Third IFIP/INWG Workshop on Protocol Specification, Verification and Testing*, Zurich, Switzerland, 1983.

[Pri '83] J.W. Prins, Master's Thesis, Delft University of Technology, Formal *description of an electronic message transfer system*, AVS Laboratory, dK/82/3, Jan. 1983. (In Dutch).

[Ra '80] R.R. Razouk, and G. Estrin, Modeling and verification of communication protocols in SARA: the X.21 interface. *IEEE Trans. On Computers*, Vol. C-29, No. 12, Dec. 1980, pp. 1038-1052.

[Rey '83] J.C.M. den Reyer, Master's Thesis, Delft University of Technology, *User aspects of an electronic message transfer system*, AVS Laboratory, dK/82/4, Sept. 1983. (In Dutch).

[Ru&We '83] H. Rudin, and C. West, (Eds.) *Proc. Third Int. Workshop on Protocol Specification, Verification and Testing*, IFIP, INWG, Zurich, Switzerland, May-June 1983.

[Sch '81] D. Schwabe, Formal specification and verification of a connection establishment protocol. *Proc. Seventh Data Comm. Symposium*, Mexico City, Oct. 1981, (IEEE), pp. 11-26.

[Su '82] C. A. Sunshine, and D.A. Smallberg, *Automated protocol verification*, USC/ISI report RR-83-110, October 1982.

[We '78] C.H. West, and P. Zafiropulo, Automated validation of a communications protocol: the CCITT X.21 recommendation, *IBM Journal of R&D*, Vol. 22, No. 1, Jan. 1978, pp. 60-71.

## APPENDIX A: SYNTAX RULES

The following defines the syntax of the Pandora protocol specification meta-language. Names given in uppercase (PROC) and quoted strings ('::'), are keywords: "tokens" to the lexical analyzer. The uppercase is used for clarity in this overview; it is optional in actual specifications.

The rules are listed in alphabetical order. The term on the left-hand side of each rule is defined by the right-hand side of the same rule. Alternatives on the right-hand side of a rule are separated by vertical bars: '|'.

A complete and syntactically correct specification should conform to the description for "prot_spec."

```
tokens: IF FI DO OD PROC REF TIMEOUT DEFAULT GOTO BREAK SKIP END

prot_spec:     module_specs '.'

annotation     /* ... any string ... */
comment        annotation | includedcode
cycle:         DO options OD
flag:          '::'
includedcode:  %% ... any C code ... %%
jump:          GOTO labelname | BREAK
labelname, msgname, procname, taskname:
               'text string'
module_spec:   task_spec | proc_spec | comment
module_specs:  module_spec | module_specs separator module_spec
options:       oneoption | oneoption options
proc_spec:     PROC procname sequence END
recv:          TIMEOUT | DEFAULT | procname '?' msgname
select:        IF options FI
send:          SKIP | procname '!' msgname
separator:     '->' | ';'
sequence:      stmnt | sequence separator stmnt
stmnt:         select | cycle | send | recv | taskname
               | labelname ':' stmnt | jump | comment
task_spec:     REF procname ':' taskname sequence END
```

A "select" statement, for instance, should be written as a keyword IF followed by a set of "options" and should be terminated by the keyword FI. Two different symbols can be used as statement separators: the traditional semi-colon ';' and the arrow '->'. The latter symbol is often used in "select" and "cycle" statements to separate incoming messages from the corresponding responses.

## APPENDIX B: INITIAL LISTING of the P2 PROTOCOL

The following lists the protocol specification that is used as a starting point for our analysis of the P2 draft protocol in section 5.

```
/*
**     The P2 draft protocol modeled in
**     the Pandora Specification Language
**/

    proc userenvironment
    do
    :: sendandnumber!sendrequest
    :: receivingUAE!receiverequest
    :: uprobe!uproberequest
    :: receivingUAE?receiveconfirmation
    :: sendandnumber?sendconfirmation
    :: uprobe?uprobeconfirmation
    :: status?statusindication
    :: receivingUAE?availableindication
    od
    end;

    proc sendandnumber
    do
    :: userenvironment?sendrequest ->
         if
         :: userenvironment!sendconfirmation
         :: userenvironment!sendconfirmation -> sendingUAE!signal1
         fi
    od
    end;

    proc sendingUAE
    do
    :: sendandnumber?signal1;
         if
         :: storeID!signal2 -> storeID?signal3:1
         :: skip
         fi;
         if
         :: submit!signal4
         :: submit!signal5
         fi
    od
    end;

    proc storeID
    do
    :: sendingUAE?signal2 -> sendingUAE!signal3:1
    :: submit?signal8
    :: uprobe?signal12
    :: notify?signal15 -> notify!signal16
    od
    end;

    ref submit : submit1
    mtl!submitrequest;
    TIM1!signal6;
    if
    :: mtl?submitconfirmation:1 ->
             TIM1!signal9;
             if
             :: skip
             :: status!signal7
             :: storeID!signal8
             :: storeID!signal8 -> status!signal7
             fi
    :: TIM1?signal10:1 ->
             if
             :: status!signal7
             :: skip
             fi
    fi
end;

ref submit : submit2
    mtl!submitrequest;
    TIM1!signal6;
    if
    :: mtl?submitconfirmation:1 ->
             TIM1!signal9
    :: TIM1?signal10:1
    fi
end;

proc submit
do
:: sendingUAE?signal4 -> submit1
:: sendingUAE?signal5 -> submit1 ; submit1
:: autoforward?signal20 -> submit2
:: autoforward?signal21 -> submit2
:: receivingUAE?signal23 -> submit2
od
end;

proc deliver
do
:: mtl?deliverindication ->
      if
      :: skip
      :: receivingUAE!signal22
      :: status!signal18
      :: autoforward!signal19
      fi
od
end;

proc autoforward
do
:: deliver?signal19 -> submit!signal20;
     if
     :: submit!signal21
     :: skip
     fi
od
end;

proc receivingUAE
do
:: deliver?signal22 -> userenvironment!availableindication
:: userenvironment?receiverequest ->
          userenvironment!receiveconfirmation;
          if
          :: submit!signal23
          :: skip
          fi
od
end;

proc notify
do
:: mtl?notifyindication -> storeID!signal15
:: storeID?signal16 -> status!signal17
od
end;

proc status
do
::   if
     :: deliver?signal18
     :: notify?signal17
     :: submit?signal7
     fi;
     userenvironment!statusindication
od
end;

proc uprobe
do
:: userenvironment?uproberequest -> mtl!proberequest;
      TIM2!signal11;
      if
      :: mtl?probeconfirmation:1 ->
              if
              :: storeID!signal12
              :: skip
              fi;
           TIM2!signal13 -> userenvironment!uprobeconfirmation
      :: TIM2?signal14:1 -> userenvironment!uprobeconfirmation
      fi
od
end;

proc TIM1
do
:: submit?signal6 ->
      do
      :: timeout -> submit!signal10:1; break
      :: submit?signal9 -> break
      :: submit?signal6
      od
:: submit?signal9
od
end;

proc TIM2
do
:: uprobe?signal11 ->
      do
      :: timeout -> uprobe!signal14:1; break
      :: uprobe?signal13 -> break
      :: uprobe?signal11
      od
:: uprobe?signal13
od
end;

proc mtl
do
:: deliver!deliverindication
:: notify!notifyindication
:: submit?submitrequest -> submit!submitconfirmation:1
:: uprobe?proberequest -> uprobe!probeconfirmation:1
od
end.
```