USING THE VALIDATOR 14

14.1 INTRODUCTION

It is time to put the tools we have developed in the last three chapters to use. First, to get our feet wet, let us look at two simple examples. The first is a reconstruction of a protocol used on the optical telegraphs in 1794 (see Chapter 1). The second is a small, but very important, example from Chapter 5: Dekker's algorithm for providing two competing processes mutually exclusive access to a critical section in their code.

14.2 AN OPTICAL TELEGRAPH PROTOCOL

The details of the communications protocols used on the optical telegraphs built in the late 18th century are hard to find. The best source is a booklet published by the Swedish inventor of a shutter telegraph Edelcrantz [1796], which comes complete with coding tables and elaborate, informal descriptions of the required coding and signaling methods. All stations along a line, except the first and the last one, had to monitor two neighboring stations for incoming traffic. Two telegraph operators were therefore usually on duty. In the validation model we build for the optical telegraph we will therefore also use two asynchronous processes, one to model the actions of each operator.

To transfer a message, the sending operator had to set the telegraph on his station to a special `start` signal, which had to be confirmed with an `attention` signal from the receiving station. The start signal could then be removed, and the first message transfered. Each message had to be reproduced faithfully by the receiver before the sender could remove it from the telegraph. (Edelcrantz system also allowed for the use of a special `error` signal, but we will not model that here.) The end of a message was signaled with a special `stop` signal. After the `stop` signal was transfered, the telegraph was released for other traffic, for instance to traffic flowing in the opposite direction.

Clearly, an operator could not use the telegraph on his station for incoming or

outgoing traffic if his colleague was already using it. We model the state of the telegraphs with a boolean array `busy[N]`, where N is the number of telegraph stations. The validation model below puts three stations in a ring (it's unlikely that they were ever used that way), with two operators per station this gives a total of six processes.

```
 1 #define true        1
 2 #define false       0
 3
 4 bool busy[3];
 5
 6 chan   up[3] = [1] of { byte };
 7 chan down[3] = [1] of { byte };
 8
 9 mtype = { start, attention, data, stop }
10
11 proctype station(byte id; chan in, out)
12 {   do
13     :: in?start ->
14             atomic { !busy[id] -> busy[id] = true };
15             out!attention;
16             do
17             :: in?data -> out!data
18             :: in?stop -> break
19             od;
20             out!stop;
21             busy[id] = false
22     :: atomic { !busy[id] -> busy[id] = true };
23             out!start;
24             in?attention;
25             do
26             :: out!data -> in?data
27             :: out!stop -> break
28             od;
29             in?stop;
30             busy[id] = false
31     od
32 }
33
34 init {
35     atomic {
36             run station(0, up[2], down[2]);
37             run station(1, up[0], down[0]);
38             run station(2, up[1], down[1]);
39
40             run station(0, down[0], up[0]);
41             run station(1, down[1], up[1]);
42             run station(2, down[2], up[2])
43     }
44 }
```

If we run a random simulation on this protocol we quickly find a problem.

```
$ spin -r -s optical
proc  6 (station)       line  23, Send start    -> queue 3 (out)
proc  5 (station)       line  23, Send start    -> queue 2 (out)
proc  4 (station)       line  23, Send start    -> queue 1 (out)
proc  3 (station)       line  13, Recv start    <- queue 2 (in)
proc  2 (station)       line  13, Recv start    <- queue 1 (in)
proc  1 (station)       line  13, Recv start    <- queue 3 (in)

#processes: 7
proc  6 (station)       line 24 (state 19)
proc  5 (station)       line 24 (state 19)
proc  4 (station)       line 24 (state 19)
proc  3 (station)       line 14 (state 4)
proc  2 (station)       line 14 (state 4)
proc  1 (station)       line 14 (state 4)
proc  0 (_init) line 44 (state 8)
7 processes created
```

The simulation gets stuck after all three stations simultaneously send out the `start` message. The three messages are received, but then the deadlock trap closes. Three operators are waiting for a confirmation of their `start` messages, the other three are waiting for the telegraph to be released by their colleagues before they can sent the required `attention` signal. In the deadlock state, three processes are at line 14 and the other three at line 24 in the source of `proctype station`.

The deadlock problem is a curious variant of Dijkstra's well-known *dining philosophers'* problem.

14.3 DEKKER´s ALGORITHM

To build a useful validation model, we extend Dekker's algorithm with two boolean variables, `ain` and `bin`, as follows:

```
 1 #define true        1
 2 #define false       0
 3 #define Aturn       false
 4 #define Bturn       true
 5
 6 bool x, y, t;
 7 bool ain, bin;
 8
 9 proctype A()
10 {   x = true;
11     t = Bturn;
12     (y == false || t == Aturn);
13     ain = true;
14     assert(bin == false);   /* critical section */
15     ain = false;
16     x = false
17 }
18
```

```
19 proctype B()
20 {   y = true;
21     t = Aturn;
22     (x == false || t == Bturn);
23     bin = true;
24     assert(ain == false);   /* critical section */
25     bin = false;
26     y = false
27 }
28
29 init
30 {   run A(); run B()
31 }
```

The variables `ain` and `bin` are set to `true` only when process `A()` or `B()`, respectively, enters its critical section. A simple `assert()` statement can be used to verify that both processes cannot be in their critical sections at the same time.

First, let us do a random simulation. The above validation model is stored in a file named ‘‘dekker.’’ We try

```
$ spin dekker
3 processes created
```

No assertion violations are reported, but the run is not very informative. We try again, this time printing out all statements.

```
$ spin -p dekker
proc  0 (_init) line 31 (state 2)
proc  1 (A)     line 11 (state 2)
proc  0 (_init) line 31 (state 3)
proc  2 (B)     line 21 (state 2)
proc  1 (A)     line 12 (state 3)
proc  2 (B)     line 22 (state 3)
proc  1 (A)     line 13 (state 4)
proc  1 (A)     line 14 (state 5)
proc  1 (A)     line 15 (state 6)
proc  1 (A)     line 16 (state 7)
proc  1 (A)     line 17 (state 8)
proc  2 (B)     line 23 (state 4)
proc  2 (B)     line 24 (state 5)
proc  2 (B)     line 25 (state 6)
proc  2 (B)     line 26 (state 7)
proc  2 (B)     line 27 (state 8)
proc  2 (B)     terminates
proc  1 (A)     terminates
proc  0 (_init) terminates
3 processes created
```

We can repeat this a few times to gain confidence that indeed the algorithm seems to perform as advertised. But that is no proof. We can easily do an exhaustive search to establish once and for all that the algorithm is correct. First we generate and compile the analyzer.

```
$ spin -a dekker
$ cc -o pan pan.c
```

That is all there is to it; except for the exhaustive validation run itself of course.

```
$ pan
full state space search for:
        assertion violations and invalid endstates
vector 16 byte, depth reached 19, errors: 0
      81 states, stored
       0 states, linked
      36 states, matched           total:      117
hash conflicts: 0 (resolved)
(max size 2^18 states, stackframes: 3/0)

unreached in proctype _init:
        reached all 3 states
unreached in proctype B:
        reached all 8 states
unreached in proctype A:
        reached all 8 states
```

The first two lines tell us what type of validation is being performed. Since no temporal claims or progress states were defined, a basic search for assertion violations and invalid end-states is performed. The next line says that the state vector for this validation model took up 16 bytes of memory, the longest unique execution sequence was 19 steps long, and, alas, there were no errors found. A total of 81 reachable system states was logged. 36 times the symbolic executions performed by the validator returned the system to a reachable state that was analyzed before. There were no hash conflicts. If there had been any, since this is a full state space search, they would have been resolved with a linked list in the hash table. All states in all processes, finally, were found to be reachable and, implicitly, we proved that no execution sequence can violate the correctness assertions: the validator tried them all. No doubt about it, the algorithm enforces mutual exclusion.

14.4 A LARGER VALIDATION

A validation of the design of the file transfer protocol from Chapter 7 is a larger job. The complete design required us to address a large number of small problems, all of which could be solved with some degree of confidence. But having solved these subproblems our job is not done. The logical consistency of the complete design is hard to assess. All the small solutions together define the behavior of a larger composite machine that can interact with its environment in an astounding number of ways. After we complete the design, the composite machine will respond in one way or another to all the possible sequences of events that the environment can offer: the ones we had in mind when we made the initial design, and all the ones we never thought of. A protocol designer quickly learns that the second class of sequences is usually larger than the first. Our job here is to find out if, despite this, the design criteria for the protocol are met.

A full listing of the protocol model, as validated here, is given in Appendix F. If all goes well, we can either prove or disprove, for instance, that this protocol is free from deadlocks, can recover gracefully from user aborts, and reliably transmits data in the presence of transmission errors.

The full protocol contains 12 asynchronous processes and 20 message channels. The model is of a realistic complexity and provides a good test case for the applicability of our tools. It is tempting to begin by trying to perform an exhaustive validation of the complete model. A straight exhaustive validation of the model, however, runs unavoidably into the traps discussed in Chapter 11; there cannot ever be enough memory or enough time to complete it. An arbitrarily placed memory limit of 16 Mbytes, for instance, is exhausted quickly and produces the following result. The maximum search depth was guessed.

```
$ spin -a pftp                  # the full model, as listed in App. F
$ cc -DMEMCNT=24 -o  pan pan.c  # set memory bound at 2^24 bytes
$ pan -m15000                   # max search depth 15,000 steps
pan: out of memory
full statespace search for:
        assertion violations and invalid endstates
search was not completed
vector 256 byte, depth reached 7047, errors: 0
   57316 states, stored
   44880 states, linked
   76300 states, matched           total:   178496
hash conflicts: 10319 (resolved)
(max size 2^18 states, stackframes: 0/1009)

memory used: 16777241
```

The exhaustive search deteriorated into an uncontrolled partial search when it exhausted the 16 Mbytes of available memory. As argued in Chapter 11, a bit state space technique can achieve better coverage in these cases, even within stricter memory bounds. For instance, with a memory arena 8 times smaller than before, a bit state space analysis reaches approximately 40 times more states:

```
$ cc -DMEMCNT=21 -DBITSTATE -o pan pan.c  # 8 times less memory
$ pan -w22 -m15000      # 2^22 = 4 Mbit = 0.5 Mbyte state space
bit state space search for:
        assertion violations and invalid endstates
vector 256 byte, depth reached 14,999, errors: 0
 2136023 states, stored
 1987936 states, linked
 3499761 states, matched           total:  7623720
hash factor: 1.963603 (best coverage if >100)
(max size 2^22 states, stackframes: 0/2365)

memory used: 1507425    # state space + 15,000 slot stack
unreached in proctype _init:
        reached all 13 states
```

```
unreached in proctype data_link:
        line 20 (state 14)
        reached: 13 of 14 states
unreached in proctype fc:
        ...
        reached: 61 of 73 states
unreached in proctype fserver:
        line 29 (state 30)
        reached: 29 of 30 states
unreached in proctype session:
        ...
        reached: 96 of 99 states
unreached in proctype present:
        ...
        reached: 32 of 34 states
unreached in proctype userprc:
        reached all 17 states
```

The analyzer inspected 7.6 million composite system states, of which more than 2 million were distinct. The state descriptions were 256 bytes long. There are, however, a number of indications that the analysis was incomplete.

- The hash factor is too low. The hash factor must be over a hundred, before we can be confident of sufficient coverage (Chapter 13).
- The depth limit of 15,000 steps was too small (note the depth-reached of 14,999 steps). The search would have to be repeated with a larger depth limit to avoid truncation.
- The list of unreached code, abbreviated above, shows that not all parts of the model were exercised.

We can boost the coverage a little bit by picking a larger memory arena, but the results are not encouraging:

```
$ cc -DMEMCNT=23 -DBITSTATE -o pan pan.c        # use more memory
$ pan -w25 -m45000               # allow up to 32 million states
bit state space search for:
        assertion violations and invalid endstates
vector 256 byte, depth reached 36569, errors: 0
18302437 states, stored
19482180 states, linked
33989843 states, matched           total: 71774460
hash factor: 1.833331 (best coverage if >100)
(max size 2^25 states, stackframes: 0/6167)

memory used: 6857209
...
```

This time, in less than half the memory arena of the first, ‘‘full search’’ we analyzed over 300 times more states using the supertrace algorithm. Still, however, the indications are that the coverage is poor. If we want to do better, we have to take a different approach. Rather than performing a single monolithic test of all layers at the same time, we can break up the validation problem into smaller, more manageable pieces. (See also the discussion of complexity management techniques such as reduction and

generalization in Chapters 8 and 11.) In the design phase we already made an effort to separate orthogonal issues, such as error control, flow control, and session control. This effort can pay off now. The correctness of the flow control layer, for instance, is completely independent of the correctness of the session control layer. We can therefore reduce the complexity of the validation substantially by validating protocol modules separately.

> *Design by stepwise refinement and validation by stepwise abstraction are complementary techniques.*

Each separate validation can achieve a much better coverage than a monolithic validation of all layers put together.

Let's look at the layers one by one. The correctness of the error control depends on the accuracy of the checksumming method, which was discussed in Chapter 3. Validation of a checksum algorithm by exhaustive reachability analysis would be inappropriate; it is a mere computation. We look at the validation of the core protocol layers: flow control, session control, and presentation. We base the validation on the assumptions that were made earlier about the behavior of the three environment processes: the user, the file server, and the data link.

14.5 FLOW CONTROL VALIDATION

The main correctness requirement for the flow control layer is that it cannot lose or reorder messages, despite the fact that the lower protocol module does lose messages. In Chapter 7 we expressed a correctness of the flow control layer, using a labeling of messages with three colors, `red`, `white`, and `blue`. To perform the validation we use the test sender and receiver process described in Chapter 7, extended with some extra code. Before any data are transferred, the test sender must synchronize the two flow control layer processes. The code is borrowed from the original session layer (see Chapter 7 and Appendix F).

```
proctype test_sender(bit n)
{       byte par, toggle;

        ses_to_flow[n]!sync,toggle;
        do
        :: flow_to_ses[n]?sync_ack,par ->
                if
                :: (par != toggle)
                :: (par == toggle) -> break
                fi
        :: timeout ->
                ses_to_flow[n]!sync,toggle
        od;
        toggle = 1 - toggle;
        do
        :: ses_to_flow[n]!white
        :: ses_to_flow[n]!red -> break
        od;
```

```
        do
        :: ses_to_flow[n]!white
        :: ses_to_flow[n]!blue -> break
        od;
        do
        :: ses_to_flow[n]!white
        :: break
        od
}
proctype test_receiver(bit n)
{
        do
        :: flow_to_ses[n]?white
        :: flow_to_ses[n]?red -> break
        :: flow_to_ses[n]?blue -> assert(0)
        od;
        do
        :: flow_to_ses[n]?white
        :: flow_to_ses[n]?red -> assert(0)
        :: flow_to_ses[n]?blue -> break
        od;
end:    do
        :: flow_to_ses[n]?white
        :: flow_to_ses[n]?red -> assert(0)
        :: flow_to_ses[n]?blue -> assert(0)
        od
}
```

The last cycle in the receiver was labeled as an end-state. It is where we would expect the receiver process to be in all valid end-states of the system. It is not wise to rely on the system reaching a deadlock state when an incorrect message is received. The receiver process blocks on unspecified receptions, but the other processes may continue, e.g., with retransmissions. For this reason, an explicit assertion violation is forced in the above validation model.

This test sender and receiver model the upper protocol layer for the flow control layer process. The lower protocol layer is the data link. It was modeled as follows:

```
proctype data_link()
{   byte type, seq;

end:    do
        :: flow_to_dll[0]?type,seq ->
                if
                :: dll_to_flow[1]!type,seq
                :: skip /* lose */
                fi
        :: flow_to_dll[1]?type,seq ->
                if
                :: dll_to_flow[0]!type,seq
                :: skip /* lose */
                fi
```

```
        od
}
```

The only function of the data link model is to simulate the loss of messages. There is, however, an equivalent and simpler way to model the same behavior. We can connect the two flow control processes directly and modify them to randomly discard any messages that arrive. This reduction allows us to remove two processes and two message channels from the model by the addition of just one clause to the receiver part of the flow control layer process (see Appendix F).

```
#if LOSS
        :: err_to_flow[N]?type,m /* lose any message */
#endif
```

We have used a preprocessor directive `LOSS` to enable or disable the possibility of message loss in validations. (The message is received, but not responded to.) In the flow control layer validation model listed in Appendix F there is one other preprocessor directive, named `DUPS`. It can be used to model the possibility of duplicate messages by triggering premature retransmissions, i.e., the retransmission of messages that are not really lost. Another step in our effort to reduce the complexity of the validation can be to group code into atomic statements wherever we can safely do so, and to combine the test sender and receiver into a single upper level tester. (See incremental composition, discussed in Chapters 8 and 11.) The complete code for the upper tester then looks as follows:

```
 1 proctype upper()
 2 {   byte s_state, r_state;
 3     byte type, toggle;
 4
 5     ses_to_flow[0]!sync,toggle;
 6     do
 7     :: flow_to_ses[0]?sync_ack,type ->
 8             if
 9             :: (type != toggle)
10             :: (type == toggle) -> break
11             fi
12     :: timeout ->
13             ses_to_flow[0]!sync,toggle
14     od;
15     toggle = 1 - toggle;
16
17     do
18     /* sender */
19     :: ses_to_flow[0]!white,0
20     :: atomic {
21             (s_state == 0 && len (ses_to_flow[0]) < QSZ) ->
22             ses_to_flow[0]!red,0 ->
23             s_state = 1
24        }
```

```
25      :: atomic {
26              (s_state == 1 && len (ses_to_flow[0]) < QSZ) ->
27              ses_to_flow[0]!blue,0 ->
28              s_state = 2
29         }
30      /* receiver */
31      :: flow_to_ses[1]?white,0
32      :: atomic {
33              (r_state == 0 && flow_to_ses[1]?[red]) ->
34              flow_to_ses[1]?red,0 ->
35              r_state = 1
36         }
37      :: atomic {
38              (r_state == 0 && flow_to_ses[1]?[blue]) ->
39              assert(0)
40         }
41      :: atomic {
42              (r_state == 1 && flow_to_ses[1]?[blue]) ->
43              flow_to_ses[1]?blue,0;
44              break
45         }
46      :: atomic {
47              (r_state == 1 && flow_to_ses[1]?[red]) ->
48              assert(0)
49         }
50      od;
51  end:
52      do
53      :: flow_to_ses[1]?white,0
54      :: flow_to_ses[1]?red,0 -> assert(0)
55      :: flow_to_ses[1]?blue,0 -> assert(0)
56      od
57  }
```

The structure of the test system we have described is shown in Figure 14.1.

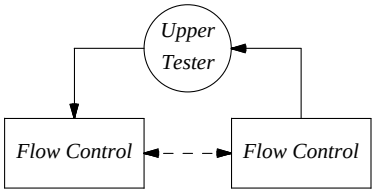

Figure 14.1 — Validation of the Flow Control Layer

The circle represents the upper level model that was added specifically for this validation. The two boxes are the flow control layer processes being validated. By the construction of the upper tester we know that if there is any error in the flow control layer, the upper tester module will trip on a false assertion.

IDEAL CHANNELS

In a first validation run we check that in the absence of errors, data are transferred correctly and the temporal claim cannot be violated. The startup script looks as follows:

```
 1 /*
 2  * PROMELA Validation Model
 3  * FLOW CONTROL LAYER VALIDATION
 4  */
 5
 6 #define LOSS                0       /* message loss   */
 7 #define DUPS                0       /* duplicate msgs */
 8 #define QSZ         2       /* queue size     */
 9
10 mtype = {
11     red, white, blue,
12     abort, accept, ack, sync_ack, close, connect,
13     create, data, eof, open, reject, sync, transfer,
14     FATAL, NON_FATAL, COMPLETE
15     }
16
17 chan ses_to_flow[2] = [QSZ] of { byte, byte };
18 chan flow_to_ses[2] = [QSZ] of { byte, byte };
19 chan dll_to_flow[2] = [QSZ] of { byte, byte };
20 chan flow_to_dll[2];
21
22 #include "flow_cl"
23 #include "upper_tester"
24
25 init
26 {
27     atomic {
28       flow_to_dll[0] = dll_to_flow[1];
29       flow_to_dll[1] = dll_to_flow[0];
30       run fc(0); run fc(1);
31       run upper()
32     }
33 }
```

The include files contain the model definitions we have just discussed. The flow control layer processes are directly linked with the first two assignments in the initial process, and they are started in the two subsequent `run` statements. The following numbered listing of the flow control layer, as tested, is useful for cross referencing the unreachable code.

```
 1 /*
 2  * Flow Control Layer Validation Model
 3  */
 4
 5 #define true        1
 6 #define false       0
 7
 8 #define M   4       /* range sequence numbers   */
 9 #define W   2       /* window size: M/2         */
10
11 proctype fc(bit n)
12 {   bool    busy[M];        /* outstanding messages    */
13     byte    q;              /* seq# oldest unacked msg */
14     byte    m;              /* seq# last msg received  */
15     byte    s;              /* seq# next msg to send   */
16     byte    window;         /* nr of outstanding msgs  */
17     byte    type;           /* msg type                */
18     bit     received[M];    /* receiver housekeeping   */
19     bit     x;              /* scratch variable        */
20     byte    p;              /* seq# of last msg acked  */
21     byte    I_buf[M], O_buf[M];     /* message buffers */
22
23     /* sender part */
24 end:        do
25     :: atomic {
26        (window < W  && len(ses_to_flow[n]) >  0
27                     && len(flow_to_dll[n]) < QSZ) ->
28                     ses_to_flow[n]?type,x;
29                     window = window + 1;
30                     busy[s] = true;
31                     O_buf[s] = type;
32                     flow_to_dll[n]!type,s;
33                     if
34                     :: (type != sync) ->
35                             s = (s+1)%M
36                     :: (type == sync) ->
37                             window = 0;
38                             s = M;
39                             do
40                             :: (s > 0) ->
41                                     s = s-1;
42                                     busy[s] = false
43                             :: (s == 0) ->
44                                     break
45                             od
46                     fi
47             }
48     :: atomic {
49             (window > 0 && busy[q] == false) ->
50             window = window - 1;
51             q = (q+1)%M
52        }
53 #if DUPS
54     :: atomic {
```

```
55              (len(flow_to_dll[n]) < QSZ
56               && window > 0 && busy[q] == true) ->
57              flow_to_dll[n]! O_buf[q],q
58         }
59 #endif
60      :: atomic {
61              (timeout && len(flow_to_dll[n]) < QSZ
62               && window > 0 && busy[q] == true) ->
63              flow_to_dll[n]! O_buf[q],q
64         }
65
66      /* receiver part */
67 #if LOSS
68      :: dll_to_flow[n]?type,m /* lose any message */
69 #endif
70      :: dll_to_flow[n]?type,m ->
71              if
72              :: atomic {
73                      (type == ack) ->
74                      busy[m] = false
75                 }
76              :: atomic {
77                      (type == sync) ->
78                      flow_to_dll[n]!sync_ack,m;
79                      m = 0;
80                      do
81                      :: (m < M) ->
82                              received[m] = 0;
83                              m = m+1
84                      :: (m == M) ->
85                              break
86                      od
87                 }
88              :: (type == sync_ack) ->
89                      flow_to_ses[n]!sync_ack,m
90              :: (type != ack && type != sync && type != sync_ack)->
91                      if
92                      :: atomic {
93                              (received[m] == true) ->
94                                      x = ((0<p-m   && p-m<=W)
95                                      ||   (0<p-m+M && p-m+M<=W)) };
96                                      if
97                                      :: (x) -> flow_to_dll[n]!ack,m
98                                      :: (!x) /* else skip */
99                                      fi
100                     :: atomic {
101                             (received[m] == false) ->
102                                     I_buf[m] = type;
103                                     received[m] = true;
104                                     received[(m-W+M)%M] = false
105                      }
106                     fi
107             fi
108     :: (received[p] == true && len(flow_to_ses[n])<QSZ
109                             && len(flow_to_dll[n])<QSZ) ->
```

```
110             flow_to_ses[n]!I_buf[p],0;
111             flow_to_dll[n]!ack,p;
112             p = (p+1)%M
113     od
114 }
```

Not knowing anything about the complexity of the model that we have constructed for the validation, the best approach is to run a quick supertrace (bit state space) analysis and check the hash factor and the number of reachable states. By multiplying the number of states stored with the number of bytes required per state we can then get an estimate of the amount of memory that would be required for an exhaustive search. For instance, a supertrace analysis of the flow control layer validation model from Figure 14.1 is performed as follows, using a memory arena of roughly 4.5 Mbytes:

```
$ spin -a pftp.flow
$ cc -DMEMCNT=23 -DBITSTATE -o pan pan.c
$ pan -w25
bit statespace search for:
        assertion violations and invalid endstates
vector 128 byte, depth reached 3781, errors: 0
   90843 states, stored
  317124 states, linked
  182422 states, matched           total:   590389
hash factor: 369.363216 (best coverage if >100)
(max size 2^25 states, stackframes: 0/418)

memory used: 4463832
...
```

The search was of good quality (the hash factor is high) so the number of states reached should be a good approximation of the true number of reachable states in the full state space. A quick calculation shows that we would need 90843×128, or roughly 12 Mbytes to store the complete state space. Having a machine with 64 Mbytes available, we can decide to repeat the analysis with an exhaustive check.

```
$ cc -DMEMCNT=24 -o pan pan.c   # memory bound 2^24
$ pan -w16                      # hash table of 2^16 slots
full state space search for:
        assertion violations and invalid endstates
vector 128 byte, depth reached 5580, errors: 0
   90845 states, stored
  317134 states, linked
  182425 states, matched           total:   590404
hash conflicts: 154271 (resolved)
(max size 2^16 states, stackframes: 0/418)

memory used: 12886356
unreached in proctype _init:
        reached all 7 states
```

```
unreached in proctype upper:
        line 13 (state 9)
        line 39 (state 29)
        line 48 (state 36)
        line 54 (state 43)
        line 55 (state 45)
        line 57 (state 49)
        reached: 43 of 49 states
unreached in proctype fc:
        line 63 (state 28)
        line 93 (state 50)
        line 96 (state 53)
        line 95 (state 55)
        line 113 (state 73)
        reached: 68 of 73 states
```

The state space built held 90,845 reachable system states, with 317,134 linked states (intermediate states in atomic sequences), and a longest unique execution sequence of 3781 steps. A total of 182,425 times a state was reached that was previously analyzed in the depth first search. The earlier bit state space analysis had 99.997% coverage.

Next, let us consider the states that are reported to be unreachable. Four of the six unreachable states in the upper tester correspond to the assertion violations that we *want* to be unreachable: lines 39, 48, 54, and 55. Line 13 specifies the action to be taken if a timeout occurs while the upper tester is waiting for a response to its initial `sync` message. It is readily checked that indeed this code should also be unreachable: if there is no message loss, the timeout should never occur. Line 57, finally, is the normal stop state of the upper tester, at the end of its code. Since the code for the upper tester is written as an infinite loop, we would also not expect that state to be reachable.

Five states are reported to be unreachable in the flow control layer protocol. The unreached code tells us that no timeout's can occur (line 63). This is correct, in the absence of message loss timeouts are redundant. It also confirms that, in the absence of all errors, acknowledgments always arrive in the exact order in which the data messages are sent (lines 93-97). Line 113, finally, is the normal end-state of the flow control layer process. Since the process never terminates, it is also correctly labeled as unreachable.

In examining the listings, remember that the line numbers are approximate, off-by-one errors are sometimes hard to avoid. In case of doubt, the state numbers given in parentheses can be used to look up the precise statement of the process in the file `pan.m`.

In the absence of message loss in the underlying data link, then, the flow control layer meets its correctness requirements. Since the assertions in the temporal claim cannot be violated, no messages can ever be lost or reordered.

MESSAGE LOSS AND DUPLICATION ERRORS

In the next validation runs we check the working of the flow control layer in the presence of two different types of errors: message loss and duplicate messages. First we check for message loss by giving the preprocessor directive `LOSS` a non-zero value. It is just within the reach of a full state space analysis.

```
$ spin -a pftp.flow1
$ cc -o pan pan.c
$ pan -w20
full state space search for:
        assertion violations and invalid endstates
vector 128 byte, depth reached 4421, errors: 0
  396123 states, stored
 1046768 states, linked
  748273 states, matched           total:  2191164
hash conflicts: 186761 (resolved)
(max size 2^20 states, stackframes: 0/543)

unreached in proctype _init:
        reached all 7 states
unreached in proctype upper:
        line 39 (state 29)
        line 48 (state 36)
        line 54 (state 43)
        line 55 (state 45)
        line 57 (state 49)
        reached: 44 of 49 states
unreached in proctype fc:
        line 113 (state 74)
        reached: 73 of 74 states
```

The timeout option in the upper tester has now been exercised, and all states of the flow control layer process were reached. All remaining unreachable states in the upper tester correspond to the error states that should be unreachable.

A next test is for duplicate messages. We enable this test with the preprocessor directive `DUPS`. This type of error dramatically increases the complexity of the model. A validation is now solidly outside the range of exhaustive searches. Only a bit state space search can still be performed with reasonable coverage.

```
$ spin -a pftp.flow2
$ cc -DMEMCNT=27 -DBITSTATE -o pan pan.c
$ pan -w29 -m100000
vector 128 byte, depth reached 56089, errors: 0
 8241456 states, stored
22946550 states, linked
21143649 states, matched           total: 52331655
hash factor: 65.142718 (best coverage if >100)
(max size 2^29 states, stackframes: 0/7621)

memory used: 70073429
unreached in proctype _init:
        reached all 7 states
```

```
unreached in proctype upper:
        line 13 (state 9)
        line 39 (state 29)
        line 48 (state 36)
        line 54 (state 43)
        line 55 (state 45)
        line 57 (state 49)
        reached: 43 of 49 states
unreached in proctype fc:
        line 63 (state 31)
        line 113 (state 76)
        reached: 74 of 76 states
```

Storing a full state space of 8,241,456 states of 128 bytes each would take a Gigabyte of memory. The bit state space search above used 70 Mbytes and completed with a hash factor of 65, thus with a reasonable guarantee of complete coverage (see Chapter 13). The longest unique execution sequence has now grown to 56,089 steps. All protocol states except those corresponding to errors and retransmission timeouts have been exercised. The flow control layer passes also this test, that is, in the absence of the other types of errors, the flow control layer seems able to cope successfully with arbitrary amounts of duplication errors.

This validation test is, of course, a rather drastic one. Premature retransmission timeouts can occur perhaps several times during a file transfer session, but very unlikely hundreds of times or more. Many other variations of validation runs are possible. We could, for instance, reduce the complexity of the search by counting and restricting the number of duplication errors per session. We can also test for combinations of loss and duplication errors, and we could intersperse the sending of `white`, `red`, and `blue` messages with flow control resynchronizations. We consider just one variant of a validation run below.

VIOLATIONS OF THE WINDOW INVARIANT

To make sure that errors are properly caught in the validation runs, we can try to tamper with the window size and replace the correct parameters:

```
#define M       4               /* range sequence numbers */
#define W       2               /* window size: M/2       */
```

in the flow control layer protocol, with, for instance

```
#define M       4               /* range sequence numbers */
#define W       3               /* window size:  > M/2    */
```

In the presence of message loss this should reveal errors, because it violates the window protocol invariant we proved earlier. We first try a search without the possibility of message loss:

```
$ spin -a pftp.flow3
$ cc -o pan pan.c
$ pan -m20000
full statespace search for:
        assertion violations and invalid endstates
vector 128 byte, depth reached 10194, errors: 0
  287445 states, stored
 1181892 states, linked
  664505 states, matched           total:  2133842
hash conflicts: 487165 (resolved)
(max size 2^18 states, stackframes: 0/1130)
```

There are more states than before, because there can be more messages outstanding at the same time, but, as expected, no errors just yet. Next, we turn on message loss by setting the compiler directive `LOSS` to 1.

```
$ spin -a pftp.flow4
$ cc -o pan pan.c
$ pan
assertion violated 0
pan: aborted (at depth 656)
pan: wrote pan.trail
full statespace search for:
        assertion violations and invalid endstates
search was not completed
vector 128 byte, depth reached 1290, errors: 1
   22469 states, stored
   45816 states, linked
   28041 states, matched           total:    96326
hash conflicts: 3267 (resolved)
(max size 2^18 states, stackframes: 0/199)
...
```

As expected, the tampering with the window protocol invariant introduces an error that is discovered in the reachability analysis after only a few thousand states are checked. It can be tracked down with a guided simulation, using the error trail produced by the analyzer.

14.6 SESSION LAYER VALIDATION

Having convinced ourselves that, with the right window size parameters, the flow control layer correctly mimics the behavior of an ideal transmission channel to the upper protocol layers, we can now use that result to simplify the validation of the session layer. We can build a validation model for this test as follows, omitting everything that was tested before:

```
/*
 * PROMELA Validation Model
 * Session Layer
 */
```

```
#include "defines2"
#include "user"
#include "present"
#include "session"
#include "fserver"

init
{       atomic {
          run userprc(0); run userprc(1);
          run present(0); run present(1);
          run session(0); run session(1);
          run fserver(0); run fserver(1);
          flow_to_ses[0] = ses_to_flow[1];
          flow_to_ses[1] = ses_to_flow[0]
        }
}
```

The session layers are connected directly, as if connected by an ideal channel that never loses, distorts or reorders messages. Since no flow control layer is present, we can comment out the code in the session layer that is specifically meant for the initialization of the flow control layer sequence numbers. The resulting code looks as follows:

```
 1 /*
 2  * Session Layer Validation Model
 3  */
 4
 5 proctype session(bit n)
 6 {   bit toggle;
 7     byte type, status;
 8
 9 endIDLE:
10     do
11     :: pres_to_ses[n]?type ->
12             if
13             :: (type == transfer) ->
14                     goto DATA_OUT
15             :: (type != transfer)   /* ignore */
16             fi
17     :: flow_to_ses[n]?type,0 ->
18             if
19             :: (type == connect) ->
20                     goto DATA_IN
21             :: (type != connect)    /* ignore */
22             fi
23     od;
24
25 DATA_IN:            /* 1. prepare local file fsrver */
26     ses_to_fsrv[n]!create;
27     do
28     :: fsrv_to_ses[n]?reject ->
29             ses_to_flow[n]!reject,0;
```

```
30              goto endIDLE
31      :: fsrv_to_ses[n]?accept ->
32              ses_to_flow[n]!accept,0;
33              break
34      od;
35                      /* 2. Receive the data, upto eof */
36      do
37      :: flow_to_ses[n]?data,0 ->
38              ses_to_fsrv[n]!data
39      :: flow_to_ses[n]?eof,0 ->
40              ses_to_fsrv[n]!eof;
41              break
42      :: pres_to_ses[n]?transfer ->
43              ses_to_pres[n]!reject(NON_FATAL)
44      :: flow_to_ses[n]?close,0 ->    /* remote user aborted */
45              ses_to_fsrv[n]!close;
46              break
47      :: timeout ->           /* got disconnected */
48              ses_to_fsrv[n]!close;
49              goto endIDLE
50      od;
51                      /* 3. Close the connection */
52      ses_to_flow[n]!close,0;
53      goto endIDLE;
54
55  DATA_OUT:           /* 1. prepare local file fsrver */
56      ses_to_fsrv[n]!open;
57      if
58      :: fsrv_to_ses[n]?reject ->
59              ses_to_pres[n]!reject(FATAL);
60              goto endIDLE
61      :: fsrv_to_ses[n]?accept ->
62              skip
63      fi;
64                      /* 2. initialize flow control *** disabled
65      ses_to_flow[n]!sync,toggle;
66      do
67      :: atomic {
68              flow_to_ses[n]?sync_ack,type ->
69              if
70              :: (type != toggle)
71              :: (type == toggle) -> break
72              fi
73         }
74      :: timeout ->
75              ses_to_fsrv[n]!close;
76              ses_to_pres[n]!reject(FATAL);
77              goto endIDLE
78      od;
79      toggle = 1 - toggle;
80                      /* 3. prepare remote file fsrver */
81      ses_to_flow[n]!connect,0;
82      if
83      :: flow_to_ses[n]?reject,0 ->
```

```
 84             ses_to_fsrv[n]!close;
 85             ses_to_pres[n]!reject(FATAL);
 86             goto endIDLE
 87     :: flow_to_ses[n]?connect,0 ->
 88             ses_to_fsrv[n]!close;
 89             ses_to_pres[n]!reject(NON_FATAL);
 90             goto endIDLE
 91     :: flow_to_ses[n]?accept,0 ->
 92             skip
 93     :: timeout ->
 94             ses_to_fsrv[n]!close;
 95             ses_to_pres[n]!reject(FATAL);
 96             goto endIDLE
 97     fi;
 98                     /* 4. Transmit the data, upto eof */
 99     do
100     :: fsrv_to_ses[n]?data ->
101             ses_to_flow[n]!data,0
102     :: fsrv_to_ses[n]?eof ->
103             ses_to_flow[n]!eof,0;
104             status = COMPLETE;
105             break
106     :: pres_to_ses[n]?abort ->      /* local user aborted */
107             ses_to_fsrv[n]!close;
108             ses_to_flow[n]!close,0;
109             status = FATAL;
110             break
111     od;
112                     /* 5. Close the connection */
113     do
114     :: pres_to_ses[n]?abort         /* ignore */
115     :: flow_to_ses[n]?close,0 ->
116             if
117             :: (status == COMPLETE) ->
118                     ses_to_pres[n]!accept,0
119             :: (status != COMPLETE) ->
120                     ses_to_pres[n]!reject(status)
121             fi;
122             break
123     :: timeout ->
124             ses_to_pres[n]!reject(FATAL);
125             break
126     od;
127     goto endIDLE
128 }
```

The user code is:

```
 1 /*
 2  * User Layer Validation Model
 3  */
 4
 5 proctype userprc(bit n)
 6 {
 7     use_to_pres[n]!transfer;
 8     if
 9     :: pres_to_use[n]?accept -> goto Done
10     :: pres_to_use[n]?reject -> goto Done
11     :: use_to_pres[n]!abort  -> goto Aborted
12     fi;
13 Aborted:
14     if
15     :: pres_to_use[n]?accept -> goto Done
16     :: pres_to_use[n]?reject -> goto Done
17     fi;
18 Done:
19     skip
20 }
```

And, finally, the presentation layer code is:

```
 1 /*
 2  * Presentation Layer Validation Model
 3  */
 4
 5 proctype present(bit n)
 6 {   byte status, uabort;
 7
 8 endIDLE:
 9     do
10     :: use_to_pres[n]?transfer ->
11             uabort = 0;
12             break
13     :: use_to_pres[n]?abort ->
14             skip
15     od;
16
17 TRANSFER:
18     pres_to_ses[n]!transfer;
19     do
20     :: use_to_pres[n]?abort ->
21             if
22             :: (!uabort) ->
23                     uabort = 1;
24                     pres_to_ses[n]!abort
25             :: (uabort) ->
26                     assert(1+1!=2)
27             fi
28     :: ses_to_pres[n]?accept,0 ->
29             goto DONE
30     :: ses_to_pres[n]?reject(status) ->
31             if
32             :: (status == FATAL || uabort) ->
33                     goto FAIL
34             :: (status == NON_FATAL && !uabort) ->
35 progress:           goto TRANSFER
36             fi
37     od;
38 DONE:
39     pres_to_use[n]!accept;
40     goto endIDLE;
41 FAIL:
42     pres_to_use[n]!reject;
43     goto endIDLE
44 }
```

We will do a validation in two separate steps. The file server, session, and presentation layer processes are all cyclic: they should never terminate. The initial process and the user processes, however, are terminating, and once they have completed their execution, the other processes must have reached a well-defined end-state. In the first validation, therefore, we can try to make sure that the system has no reachable invalid end-states. We can do this with an exhaustive validation, as follows:

```
$ spin -a pftp.ses
$ cc -o pan pan.c
$ pan -w19
full state space search for:
        assertion violations and invalid endstates
vector 144 byte, depth reached 451, errors: 0
  509179 states, stored
       9 states, linked
  576192 states, matched           total:  1085380
hash conflicts: 369417 (resolved)
(max size 2^19 states, stackframes: 0/23)

unreached in proctype _init:
        reached all 12 states
unreached in proctype fserver:
        line 29 (state 30)
        reached: 29 of 30 states
unreached in proctype session:
        line 48 (state 37)
        line 94 (state 64)
        line 95 (state 65)
        line 124 (state 93)
        line 128 (state 99)
        reached: 94 of 99 states
unreached in proctype present:
        line 26 (state 15)
        line 44 (state 34)
        reached: 32 of 34 states
unreached in proctype userprc:
        reached all 17 states
```

The unreached code in the presentation layer (line 26) indicates that no case was found in which two subsequent `abort` messages are received from the user process. Checking the user process, we can quickly see why that is: the user process does not allow it. The unreached code in the session layer protocol, however, flags an incompleteness in this first validation test. The unreached lines 48, 94, 95, 124, and line 128 are responses to `timeout` conditions that were included to allow the session layer to recover from a sudden loss of communication with its peer process. This possibility, however, is not modeled as part of the channel behavior and cannot be exercised.

To verify also that these `timeout` conditions cannot cause havoc, we must revise the validation model. We can do so by adding a few lines to the initialization code in the `init` process given above:

```
        atomic
        {       byte any;
                chan foo = [1] of { byte, byte };
                ses_to_flow[0] = foo;
                ses_to_flow[1] = foo
        };
```

```
end:    do
        :: foo?any,any
        od
}
```

At any time after the initial start-up of the protocol, these extra lines can now be executed. The effect is that the two peer session layer processes are disconnected. The loop at the end removes all the messages that the two session layers produce. The extension increases the complexity of the test somewhat more, but a bit state space analysis is still feasible. The result is now

```
$ spin -a pftp.ses1
$ cc -DBITSTATE -o pan pan.c
$ pan -w29
bit state space search for:
        assertion violations and invalid endstates
vector 148 byte, depth reached 456, errors: 0
 1686543 states, stored
  246135 states, linked
 1960294 states, matched           total:  3892972
hash factor: 318.326063 (best coverage if >100)
(max size 2^29 states, stackframes: 0/25)

unreached in proctype _init:
        line 31 (state 19)
        reached: 18 of 19 states
unreached in proctype fserver:
        line 29 (state 30)
        reached: 29 of 30 states
unreached in proctype session:
        line 128 (state 99)
        reached: 98 of 99 states
unreached in proctype present:
        line 26 (state 15)
        line 44 (state 34)
        reached: 32 of 34 states
unreached in proctype userprc:
        reached all 17 states
```

Compared to the first test, we have now explored over three times as many states and effectively reached all relevant protocol states. The hash factor is large enough to be confident that close to 100% of the reachable system states have been tested within the memory arena that is available. An exhaustive search would have required at least 1,686,543×148 or 249 Mbytes of memory, four times more than we have used.

THE TEMPORAL CLAIM

In the second validation of the session layer protocol that we undertake here, we consider the temporal claim that was formulated in Chapter 7.

```
never {
        do
        :: !pres_to_ses[n]?[transfer]
        && !flow_to_ses[n]?[connect]
        :: pres_to_ses[n]?[transfer] ->
                goto accept0
        :: flow_to_ses[n]?[connect] ->
                goto accept1
        od;
accept0:
        do
        :: !ses_to_pres[n]?[accept]
        && !ses_to_pres[n]?[reject]
        od;
accept1:
        do
        :: !ses_to_pres[1-n]?[accept]
        && !ses_to_pres[1-n]?[reject]
        od
}
```

Since the protocol is symmetric, it suffices to validate this claim for just one value of n, e.g., zero. The result is as follows:

```
$ spin -a pftp.ses2
$ cc -o pan pan.c
$ pan
cycle of length 6 (99) 104
pan: accept state in cycle (at depth 99)
pan: wrote pan.trail
full statespace search on behavior restricted to claim for:
        assertion violations
        and absence of acceptance labels in all cycles
search was not completed
vector 148 byte, depth reached 100, errors: 1
     151 states, stored
       9 states, linked
      16 states, matched           total:      176
hash conflicts: 0 (resolved)
(max size 2^18 states, stackframes: 0/4)
```

An acceptance cycle was detected, which means that the claim can be violated. A closer look with the simulator can reveal the cause.

```
$ spin -t -r -s pftp.ses2       # -t: follow trail produced by pan
proc  3 (userprc) line   8, Send transfer -> queue 6 (use_to_pres[1])
proc  5 (present) line  11, Recv transfer <- queue 6 (use_to_pres[1])
proc  5 (present) line  19, Send transfer -> queue 4 (pres_to_ses[1])
proc  7 (session) line  11, Recv 13     <- queue 4 (pres_to_ses[1])
proc  7 (session) line  56, Sent open   -> queue 8 (ses_to_fsrv[1])
...
<<<<<START OF CYCLE>>>>>
proc  9 (fserver) line  13, Recv data   <- queue 8 (ses_to_fsrv[1])
proc  6 (session) line 101, Send data,0 -> queue 1 (ses_to_flow[0])
```

```
proc  8 (fserver) line  23, Sent data   -> queue 9 (fsrv_to_ses[0])
proc  6 (session) line 100, Recv data   <- queue 9 (fsrv_to_ses[0])
proc  7 (session) line  37, Recv data,0 <- queue 1 (flow_to_ses[1])
spin: trail ends after 179 steps
step 179, #processes: 10
...
```

The validator discovered here that the number of `data` messages that is exchanged during a file transfer session is not bounded. This means that the sending of a final `accept` or `reject` message to the presentation layer can be postponed indefinitely, which is a direct violation of our correctness requirement.

To fix this problem we can try telling the temporal claim to ignore `data` messages, that is, to consider only zero-length file transfers.

```
never {
        do
        :: !pres_to_ses[0]?[transfer]
        && !flow_to_ses[0]?[connect]
        :: pres_to_ses[0]?[transfer] ->
                goto accept0
        :: flow_to_ses[0]?[connect] ->
                goto accept1
        od;
accept0:
        do
        :: !ses_to_pres[0]?[accept]
        && !ses_to_pres[0]?[reject]
        && !ses_to_flow[0]?[data]
        od;
accept1:
        do
        :: !ses_to_pres[1]?[accept]
        && !ses_to_pres[1]?[reject]
        && !ses_to_flow[1]?[data]
        od
}
```

The validation with this new claim proceeds as follows:

```
$ spin -a pftp.ses3
$ cc -o pan pan.c
$ pan
cycle of length 5 (99) 103
pan: accept state in cycle (at depth 99)
pan: wrote pan.trail
full state space search on behavior restricted to claim for:
        assertion violations
        and absence of accept states in all cycles
search was not completed
vector 148 byte, depth reached 132, errors: 1
   21645 states, stored
       9 states, linked
   20316 states, matched           total:    41970
```

```
hash conflicts: 2293 (resolved)
(size 2^18 states, stackframes: 0/5)
```

Again, the validator discovered that the correctness requirement can be violated. The relevant part of the trail is as follows:

```
$ spin -t -r -s pftp.ses3
...
proc  4 (present) line 19, Send transfer -> queue 3 (pres_to_ses[0])
proc  6 (session) line 42, Recv transfer <- queue 3 (pres_to_ses[0])
<<<<<START OF CYCLE>>>>>
proc  6 (session) line 43, Send reject,NON_FATAL -> \
                                          queue 11 (ses_to_pres[0])
proc  4 (present) line 31, Recv reject,15 <- queue 11 (ses_to_pres[0])
proc  4 (present) line 19, Send transfer -> queue 3 (pres_to_ses[0])
spin: trail ends after 176 steps
...
```

After a file transfer has started, there can be an unbounded number of conflicting transfer requests from the remote peer process. Again, processing these requests as non-fatal rejects can postpone for arbitrarily long the sending of the final `accept` or `reject` message for the active file transfer.

This time it is much harder to modify the temporal claim to remove this pattern from consideration. An acceptance-state label identifies events as potentially bad. In this case, however, we can work more effectively with a method for labeling a small set of events as good and focus on others. The right tool for that is the progress-state label. If we can rephrase the temporal claim as a correctness requirement on the absence of non-progress cycles it becomes easier to exclude certain patterns from consideration.

Note that, if we disregard the two patterns discovered earlier, all executions of the session layer protocol must terminate. Any cycle that can be identified, therefore, will become a non-progress cycle and thus a detectable a violation of the correctness requirements. We label the states `DATA_IN` and `DATA_OUT` in the session layer protocol as progress states. To exclude the two patterns discovered above, we also label the data exchanges with the file server as progress states, plus one state in the presentation layer protocol. A new listing of the presentation layer is given below:

```
proctype present(bit n)
{       byte status, uabort;

endIDLE:
        do
        :: use_to_pres[n]?transfer ->
                uabort = 0;
                break
        :: use_to_pres[n]?abort ->
                skip
        od;

TRANSFER:
        pres_to_ses[n]!transfer;
```

```
        do
        :: use_to_pres[n]?abort ->
                if
                :: (!uabort) ->
                        uabort = 1;
                        pres_to_ses[n]!abort
                :: (uabort) ->
                        assert(1+1!=2)
                fi
        :: ses_to_pres[n]?accept ->
                goto DONE
        :: ses_to_pres[n]?reject(status) ->
progress:       if
                :: (status == FATAL || uabort) ->
                        goto FAIL
                :: (status == NON_FATAL && !uabort) ->
                        goto TRANSFER
                fi
        od;
DONE:
        pres_to_use[n]!accept;
        goto endIDLE;
FAIL:
        pres_to_use[n]!reject;
        goto endIDLE
}
```

The validation is straightforward from this point on.

```
$ spin -a pftp.ses4
$ cc -DBITSTATE -o pan pan.c
$ pan -l -w28
bit state space search for:
        assertion violations and non-progress loops
vector 148 byte, depth reached 458, non-progress loops: 0
  847134 states, stored
      18 states, linked
 1104341 states, matched           total:  1951493
hash factor: 316.874472 (best coverage if >100)
(size 2^28 states, stackframes: 0/489)
```

A bit state space analysis completed with good coverage. No non-progress cycles were discovered, which means that with good probability the correctness requirements are met.

FURTHER REDUCTIONS

To confirm the earlier results with an exhaustive validation, we could pursue several options. Incremental composition and generalization can be used to combine the user and presentation layer processes into a single environment process to the session layer. This model may look as follows, appropriately labeled with progress tags:

```
/*
 * PROMELA Validation Model
 * Presentation & User Layer - combined and reduced
 */

proctype present(bit n)
{       byte status;

progress0:
        pres_to_ses[n]!transfer ->
        do
        :: pres_to_ses[n]!abort;
progress1:      skip
        :: ses_to_pres[n]?accept,status ->
                        break
        :: ses_to_pres[n]?reject,status ->
                if
                :: (status == NON_FATAL) ->
                        goto progress0
                :: (status != NON_FATAL) ->
                        break
                fi
        od
}
```

The external behavior of this process is indistinguishable from the external behavior of the two separate processes, with one important exception: the new model is less well-behaved. The reduced model can spark an arbitrary number of `abort` messages while a transfer request is outstanding. If the session layer protocol is correct for this environment, it must also be correct with respect to the original one, simply because the original behavior is a subset of the new one. The validation can now be done exhaustively and produces the following result:

```
$ spin -a pftp.ses5
$ cc -DMEMCNT=27 -o pan pan.c
$ pan -l -m2000
full state space search for:
        assertion violations and non-progress loops
vector 132 byte, depth reached 1783, non-progress loops: 0
  553987 states, stored
       8 states, linked
  798367 states, matched           total:  1352362
hash conflicts: 990275 (resolved)
(size 2^18 states, stackframes: 0/325)

memory used: 70872461
```

The validation run confirms that the correctness requirement of the session layer protocol is properly met. Had this first reduction been insufficient, further reduction steps could still be taken to force an exhaustive validation. All interactions of the session layer with the file server, for instance, could be removed and replaced with equivalent nondeterministic choices within the session layer. Similarly, the combined

user and presentation layer could be merged into the session layer protocol to produce a single process that represents the behavior of one protocol session layer entity. The combinations can be made manually, carefully preserving the equivalence with the original model, or automatically with an incremental composition method as discussed in Chapters 8 and 11.

14.7 SUMMARY

Our admiration for programmers who can design and debug a protocol using only tools developed for sequential systems can only grow after the first experience with an automated protocol validation system. It is, of course, not really surprising that the validation runs reported in this chapter have failed to reveal serious errors in the design from Chapter 7. The errors were certainly present in the initial versions of the protocol, but were found with SPIN and removed before these final tests were performed. Most of the errors found in the earlier stages of the design were cases of incompleteness that are very hard to find by manual inspection of the code.

Given a machine of reasonable size, the basic protocols for session control and flow control can fairly easily be validated with purely exhaustive searches of all reachable system states. This much is well within the power of the automated tools. The tools are severely tested by the exception conditions that must be validated: message loss, duplication errors, and hangups. The increase in complexity makes it impossible to perform the traditional completely exhaustive validations. Bit state space hashing proves to be a powerful alternative here. As an example, one test performed for an earlier version of the session layer protocol generated 15,462,939 system states of 472 bytes each. A full state space that stores all these states would be over 7 Gigabytes (7,298,507,208 bytes), well beyond what can effectively be stored or processed. On a machine with 64 Mbytes of memory available for the search, no more than 142,179 of these states can be stored in a full state space search: a coverage of less than 1%. The bit state space technique, using the same amount of memory, can accommodate over 250,000,000 states, more than 15 times what is required. With this method we could effectively increase the coverage of that search from less than 1% to one that, with high probability, is close to 100%. No other method known to date can do better.

EXERCISES

14-1. Validate your favorite protocol with the tools described here. □

14-2. Develop and implement more specific tools for automating the generalization or incremental composition of PROMELA models (research project). □

BIBLIOGRAPHIC NOTES

A detailed validation study as performed in this chapter is rarely documented. The first automated validations were reported in West and Zafiropulo [1978], though the analytical power of our tools has grown substantially since then. The validation method applied in this chapter was originally described in Holzmann [1987b, 1988]. Its capabilities are compared with more conventional approaches to the protocol

validation problem in Holzmann [1990]. It has been applied to systems that are ordinarily well outside the range of exhaustive validation, as reported in Holzmann and Patti [1989].