A PROTOCOL VALIDATOR 13

297 Introduction 13.1
298 Structure of the Validator 13.2
299 The Validation Kernel 13.3
302 The Transition Matrix 13.4
303 Validator-Generator Code 13.5
306 Overview of the Code 13.6
308 Guided Simulation 13.7
310 Some Applications 13.8
315 Coverage in Supertrace Mode 13.9
316 Summary 13.10
316 Exercises
317 Bibliographic Notes

13.1 INTRODUCTION

To extend the protocol simulator from Chapter 12 with an validator generator, all we have to do is to activate two command line options from the source listing in Appendix D:

`-a`, To generate a protocol specific SPIN analyzer
`-t`, To follow an error trail produced by that analyzer

To do this we have to replace the two dummy routines `gensrc()` and `match_trail()` in Appendix D with real code. To see how the analyzer is used refer to Section 13.8 or Chapter 14.

The analyzer described here is based on the discussion in Chapter 11. To keep the code reasonably simple, we will not discuss a complete implementation of the state vector model. Even without that, it takes a fair amount of code to produce a validator of good performance. But once the job is done right an efficient validator can be produced in a matter of seconds, and can be applied to problems of arbitrary complexity. The validators that are produced by SPIN in this way are among the fastest programs for exhaustive searching known to date. A full implementation of the state vector model can secure a still better performance, but that is well beyond the scope of this book.

The validators can be used in two different modes. For small to medium size models the validators can be used with an exhaustive state space. The result of all validations performed in this mode is equivalent to an exhaustive proof of correctness, for the correctness requirements that were specified (by default, absence of deadlock). For systems that are larger, the validators can also be used in *supertrace mode*, with the bit state space technique as discussed in Chapter 11. In these cases the validations can be performed in much smaller amounts of memory, and still retain excellent coverage

of the state space. The results of all validations performed in supertrace mode are superior to any other type of validation performed within the same physical constraints of the host machine (e.g., memory size and speed).

To produce an analyzer, the parse tree that is constructed by the SPIN simulator is translated into a C program, and extended with state space searching modules. The program that is generated is then compiled stand-alone. When it is executed it performs the required validation. If an error is discovered, the program writes a simulation trail into a file and stops. The simulation trail can be read by the original simulator, which can then reproduce the error sequence and allow the user to probe the cause of the error in detail.

Below we first discuss the general structure of the protocol analyzers that are generated. We then give an overview of the routines that extract the protocol specific information from the SPIN parse tree. Next we discuss the extensions of the simulator to provide for guided simulations. We conclude with some examples of the usage of the new tool.

13.2 STRUCTURE OF THE VALIDATOR

Figure 13.1 shows the main components of the analyzers that can be generated.

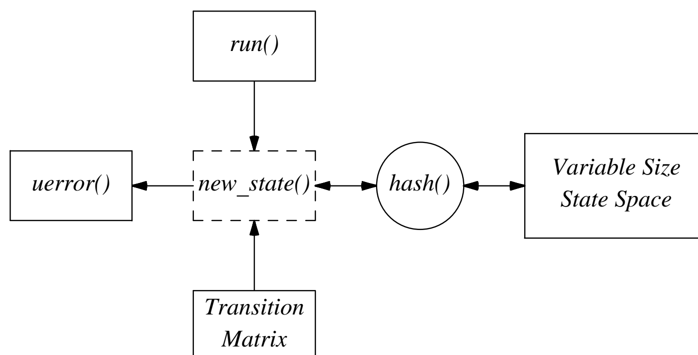

Figure 13.1 — Structure of SPIN Validators

A procedure called `run()` allocates memory and prepares all data structures that the validator will use during the search. It calls a single procedure `new_state()` to perform the actual search. The two main data structures used by this procedure `new_state()` are the state space and a large transition matrix that encodes the complete PROMELA validation model. Each statement in the model produces an entry in this matrix, defining precisely the executability predicate and the effect of the execution. Every `proctype` definition contributes entries to this matrix.

The current state of the system is maintained in a vector of values that can grow and shrink dynamically: a rubber state-vector. PROMELA `run` statements append new processes to the state-vector. The rubber state-vector, therefore, fulfills a role that is

similar to that of the run queue in the simulator scheduler. Procedure `new_state()` performs a depth first search of all executable statements in the model. Instead of selecting just one executable statement from the list of runnable processes, as the simulator did, the validator's job is to test the effects of *all* executable statements, in all possible interleavings. Before starting the analysis for a new state `new_state()` consults the state space, via the `hash()` function and decides whether the current state was analyzed before and can be skipped.

If an error is found, procedure `uerror()` is called to produce an error trail for the simulator and, unless otherwise specified, analysis stops. An error can be any violation of the formal correctness requirements, for instance, a local assertion failure or a global system state in which all processes are permanently blocked.

Finding an inconsistency in the SPIN model and assisting the user in determining its causes, is done with two different tools: validator and simulator. The rationale behind this approach is the standard UNIX discipline: each tool we develop should do one thing, and do it well.

- The simulator is designed as an interactive tool. It has a short start-up time and can give a a detailed look at the working of the protocol.
- The validator is a non-interactive tool. It has a longer start-up time, since it requires the compilation of an intermediate program, but it is optimized for exhaustive searches.

13.3 THE VALIDATION KERNEL

Procedure `new_state()` is the core of the analyzer. It controls all executions, monitors progress, and performs the correctness checks. More than half of the runtime is spend in this routine, with the larger part of the remainder being used up in the calculation of hash values to access the state space.

The procedure is statically defined in a header file named `pangen1.h`. Different portions of the code are enabled or disabled depending on the presence or absence of rendezvous communications, temporal claims, acceptance states, or progress states, and depending on the type of state space storage that is selected. Ignoring, for the time being, all these options, the plain exhaustive state searching algorithm looks as follows:

```
 1 new_state()
 2 {   register Trans *t, *ta;
 3     char n, m, ot, lst;
 4     short II, tt;
 5     short From = now.nr_pr-1;
 6     short To = 0;
 7 Down:
 8     if (depth >= maxdepth)
 9     {       truncs++;
10             goto Up;
11     }
```

```
12      if (To == 0)
13      {       if (hstore((char *)&now, vsize))
14              {       truncs++;
15                      goto Up;
16              }
17              nstates++;
18      }
19      if (depth > mreached)
20              mreached = depth;
21      n = timeout = 0;
22
23 Again:
24      for (II = From; II >= To; II -= 1)
25      {       this = pptr(II);
26              tt = (short) ((P0 *)this)->_p;
27              ot = (unsigned char) ((P0 *)this)->_t;
28              for (t = trans[ot][tt]; t; t = t->nxt)
29              {
30 #include "pan.m"
31 P999:                   /* jumps here when move succeeds */
32                      if (m>n||(n>3&&m!=0)) n=m;
33                      depth++; trpt++;
34                      trpt->pr = II;
35                      trpt->st = tt;
36                      if (t->st)
37                      {       ((P0 *)this)->_p = t->st;
38                              reached[ot][t->st] = 1;
39                      }
40                      trpt->o_t  =  t; trpt->o_n  = n;
41                      trpt->o_ot = ot; trpt->o_tt = tt;
42                      trpt->o_To = To;
43                      if (t->atom&2)
44                      {       From = To = II; nlinks++;
45                      } else
46                      {       From = now.nr_pr-1; To = 0;
47                      }
48                      goto Down;      /* pseudo-recursion */
49 Up:
50                      t = trpt->o_t; n = trpt->o_n;
51                      ot = trpt->o_ot; II = trpt->pr;
52                      tt = trpt->o_tt; this = pptr(II);
53                      To = trpt->o_To;
54 #include "pan.b"
55 R999:                   /* jumps here when done */
56                      depth--; trpt--;
57                      ((P0 *)this)->_p = tt;
58              } /* all options */
59      } /* all processes */
60
```

```
61     if (n == 0)
62     {       if (!endstate() && now.nr_pr)
63             {       if (!timeout)
64                     {       timeout=1;
65                             goto Again;
66                     }
67                     uerror("deadlock");
68             }
69     }
70     if (depth > 0) goto Up;
71 }
```

The procedure is called once and does not return until either a complete search is performed or an error found. In this version, without all the trimmings of a full implementation, the only type of error checked for is an invalid end-state. The main work is done in two `for` loops. The first one, on line 24, loops over all currently executing processes. The second one, on line 28, exhaustively checks all executable statements in each process. The `proctype` of the current process is stored in a local variable `ot`, and the process state is kept in a local variable `tt`. These two variables together are used to index the *transition matrix*, `trans[ot][tt]`, on line 28. A pointer `T` points to the definition of the transition itself: the condition, the effect and the next state. The execution of the transitions themselves are hidden in a file `pan.m` that is included on line 30. It is a simple case switch that records all transitions that are defined in the system. If a transition is executable, it leads to label P999. If it is unexecutable a `continue` is executed that brings us back into the inner loop on line 28.

A successful transition produces a new state that must be analyzed in precisely the same fashion as the current one. This is where normally a recursion step is executed. The time and space required for the recursive procedure calls, however, can easily be avoided if the recursion is replaced with iteration. Let us look at how this is implemented.

Lines 32 to 42 perform some housekeeping to prepare the validator for the analysis of a newly generated state. The depth count is increased, a pointer is incremented for the user level stack `stptr`, which maintains, among others, the execution trail. If the transition was labeled *atomic*, line 43 makes sure that the current process will continue executing in the next step, foregoing options for executions in the other processes. The default case is invoked on line 46, defining that all currently execution processes must be considered. The recursion step is replaced on line 48 with a jump to the label `Down.`

On the return from the pseudo recursion, with a jump to label `Up` on line 48, all relevant local variables are retrieved via the stack pointer `stptr`, which, if all is well, points at exactly the location where they were saved before the matching jump to `Down.` Then the state vector is restored to its original value by performing a reverse operation that undoes the effect of the last forward transition that was explored. The code that does this is hidden in a separate file `pan.b` that is included on line 48. This file also contains a case switch, that relies on the pointer to the transition matrix `t` to

point the program to the right operation to execute.

The state itself, storing state information about all currently executing processes and all currently accessible queues and variables, is maintained in a global variable `now`, though that is not visible in the body of this procedure.

The label `Again`, on line 23, with the matching jump on line 65, is used to implement the `timeout` recovery mechanism. If the search gets stuck this is first noticed on line 61 by the zero value of variable `n`. A quick check is then performed to see if the deadlock is not in fact a valid endstate. If this test fails, the timeouts are enabled and second attempt is made to perform a transition with a return to label `Again`. If this also fails, the error routine `uerror()` is called, which can trigger the writing of an error trail and optionally abort the search.

The extensions that are needed to implement the full range of correctness checks discussed in Chapter 6 triple the size of the algorithm, though in a none too exciting way. To check temporal claims, for instance, the search alternately executes atomic statements in the model and in the claim. The toggle bit, which determines where to look for the next statement to execute, is piggybacked onto the variable `tau`. To implement rendezvous message passing, a global variable `boq` (short for ''blocked on queue'') is set after every rendezvous send operation. The variable blocks all operations other than a matching receive operation. Effectively, then, the send-receive handshake becomes one indivisible step, though the validator performs it as two distinct transitions. The other extensions similarly make the algorithm somewhat harder to read, but do not change it in a fundamental way. In the discussion we therefore restrict ourselves mainly to the basic version. A listing of the complete algorithm for exhaustive validation is given in Appendix E.

13.4 THE TRANSITION MATRIX

The transition matrix shown in Figure 13.1 performs a central role in the search. Some precautions are taken to make sure that it does not contain any spurious moves that could slow down the search. It is constructed from elements of the following type:

```
typedef struct Trans {
        short atom;      /* is this an atomic transition */
        int st;          /* the next state/statement     */
        int forw;        /* index for forward transition */
        int back;        /* index for return  transition */
        struct Trans *nxt;",
} Trans;
```

Every `proctype` in the original SPIN specification defines a set of entries in the matrix: one for every control flow state. The following array is used to keep track of them.

```
Trans *trans[NPROCS][NSTATES]
```

The transitions of a process of type 19 for control flow state 90 can be found through

the pointer

```
trans[19][90]
```

Figure 13.2 illustrates a typical use of the matrix elements.

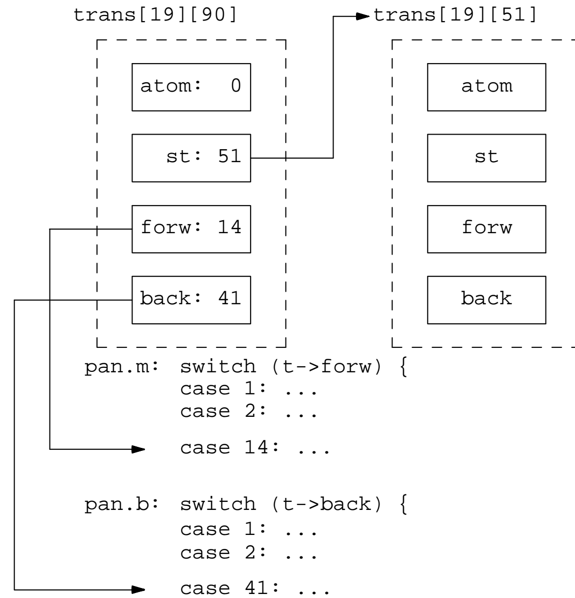

Figure 13.2 — Transition Matrix Elements

The first field of a `Trans` structure is used to label transitions that are atomic. A zero entry means that the transition is a normal, asynchronous one. Field `st` defines the successor state that is reached if a transition is successfully executed. The next field `forw` is an index that identifies the right operation to execute. It indexes a switch of transitions that is generated in the include file `pan.m`. Similarly, `back` is an index into the switch of `pan.b` that identifies the operation that can undo the effect of the forward transition and restore the state vector to its original state.

Whenever there is a nondeterministic choice of transitions to make, e.g., for SPIN selection and repetition structures, the options are placed in a linked list that is connected via the transition element's `nxt` pointer (not shown in Figure 13.2). To examine all executable options, the inner loop of `new_state()` simply walks down the linked list.

13.5 THE VALIDATOR-GENERATOR CODE

It would seem that all that is left to do to make the analyzer run is to generate the right transition matrix. Well, not quite. Since we are generating code anyway we can also generate protocol specific routines for manipulating the queues, instantiating process instances, and the like. It is a fairly substantial piece of code that generates all this information, but well worth it.

MESSAGE CHANNELS

Before discussing the code of the validator generator itself, let us look briefly at the code that it tries to produce. From the 25 relevant routines that are part of every analyzer generated, 7 deal with the message channels:

```
addqueue()
delq()
q_restor()
qsend()
qrecv()
unsend()
unrecv()
```

The first routine, for instance, implements the PROMELA `chan` statement, and the second routine removes a channel when it goes out of scope, that is, when the creating process has terminated. For every different type of message channel, a separate queue template is generated. For example,

```
typedef struct Q1 {
        unsigned char Qlen;     /* q_size */
        unsigned char _t;       /* q_type */
        struct {
                unsigned fld0 : 32;
        } contents[1];
} Q1;
```

defines a queue with one slot and one message field. Two queue fields are predefined: one specifies the type of the channel and the other the current number of messages that the corresponding queue stores. The routine `q_restor()` is used in backward moves to restore a deleted queue to its last known state, just before a deletion with `qdel()`. Messages are appended to a queue with the procedure `qsend(into, fld0, ...)`. Parameter `fld0` indicates the value of the first message parameter. Procedure `qrecv(from, slot, fld, done)` retrieves a single message field `fld` from slot `slot` in a queue `from`. The parameter `done` is set to one after all fields have been extracted and the message can be removed from the queue. Note that a single receive operation can have multiple side-effects by setting variables. The definition of a procedure that reads one message parameter at a time is a simple general solution to that problem.

Every action has an undo. The counterparts of send and receive are named `unsend()` and `unrecv()`, which, respectively, remove a complete message from the tail of a queue or put one back at its head.

PROCESSES

Three routines deal with processes.

```
addproc()
delproc()
p_restor()
```

Every type of process is again defined in a different template. For instance, for the

factorial program we find

```
typedef struct P1 {      /* factorial */
        unsigned _t : 2;        /* proctype */
        unsigned _p : 4;        /* state    */
        int result;
        unsigned char child;
        int n;
        unsigned char p;
} P1;
```

The process type and the current process state are a standard part of the template. The required width of the bitfields is calculated by the generator. These first two fields are used to index the transition matrix. The remaining entries reserve slots for local variables. The procedure `addproc()` appends one of these templates to the state vector, and `delproc()` removes it. Procedure `p_restor()` is used in backward moves to restore a deleted process to its last known state.

STATE SPACE MAINTENANCE

The state space can be accessed in two different ways, selectable by a preprocessor directive named `BITSTATE`. Unless this name has been defined, a full state space is constructed with a traditional exhaustive search method. To access the state space the routine `hstore()` is used. Seen from procedure `new_state()` this looks as follows:

```
if (hstore((char *)&now, vsize))
{       truncs++;
        goto Up;
}
nstates++;
```

The variable `vsize` gives the current size of the state vector in bytes. The global `now` points to it. If `hstore()` returns the boolean value `false`, the state is new and must be analyzed. As a side effect of `hstore()`, the state is also stored in full in the state space. The next time that this same state is encountered the routine will return the boolean value `true`, which means that the state can be skipped. Internally `hstore()` uses a fast hash function called `s_hash()`.

If the name `BITSTATE` is explicitly defined during compilation, the more memory efficient supertrace, bit state space memory routines are used. They invoke a double value hashing function called `d_hash()`. The routine uses the state vector to calculate two different hash values (see Chapter 11, page 230). A check is then made on the two bit positions in the state space, and if a double match is found, the state is assumed to have been analyzed before. This is how it works:

```
d_hash((unsigned char *) &now, vsize);
j3 = (1<<(J1&7)); j1 = J1>>3;
j4 = (1<<(J2&7)); j2 = J2>>3;
if ((SS[j2]&j3) && (SS[j1]&j4))
{       truncs++;
        goto Up;
}
SS[j2] |= j3; SS[j1] |= j4;     /* storage */
nstates++;
```

First `d_hash()` is called to produce two hash values for the first `vsize` bytes of the state vector, stored in `now`. The values are written into the integer global variables `J1`, and `J2`. The next few operations take the low order 3 bits from `J1` and `J2`, using the bit mask 7, and assign them to `j3` and `j4`. The remaining bits are shifted down by three bit positions and assigned to `j1` and `j2`. The test

```
if ((SS[j2]&j3) && (SS[j1]&j4))
```

selects the calculated bit positions and only if both bits are on a match is assumed. The last line sets the two bits with a binary OR operation, to secure a future match on the same state. It is the only storage operation performed: a savings in memory of (8×`vsize`−2) bits per state, assuming 8 bits per byte.

THE REMAINING ROUTINES

The remainder of the validation routines is fairly straightforward. There is a routine `endstate()` to determine if the current combination of process states is a valid end-state, by comparing them with the known stop states in each process. A routine `assert()` checks user defined assertions and produces an error trail if one is violated. There is a routine `r_ck()` for every process type in the system that performs the reachability check after the depth-first search by verifying that every relevant control flow state in the specification has indeed been reached by at least one of the executing processes. Two main routines deal with the transition matrix, `settable()` and `retrans()`. The first sets the table (matrix) to its default contents as produced by the generator, using the parse tree structure. The second quickly goes through the structure to optimize it a little for the validation task. Nested choices, for instance, are rewritten into single choices, without, of course, violating the semantics of PROMELA.

13.6 OVERVIEW OF THE CODE

The code for the generator is included in four C files. At the time of writing, a count of these routines produced:

```
$ wc pangen[1-4].c
    424    1341    8812 pangen1.c
    501    1784   13595 pangen2.c
    102     303    1659 pangen3.c
    170     583    4024 pangen4.c
```

Three header files contain fixed code that is included with every program generated:

```
$ wc pangen[1-2].h
    910    3363   22276 pangen1.h
    127     528    3389 pangen2.h
    108     348    2151 pangen3.h
```

And finally, one more file is used to implement the guided simulation option.

```
$ wc pangen5.c
    158     489    3292 pangen5.c
```

The complete code for the exhaustive validation option can be found in Appendix E. Here we highlight only the major parts.

The routine that is actually called by the simulator, if the command line option `-a` is given, is called `gensrc()`. It is included in `pangen2.c`. It starts by creating the five target files `pan.[chtmb]` and copying some code from `pangen2.h` into them. It then calls procedure `putproc()` once for every basic `proctype` that was parsed: once for the description of the `init` process, stored in the simulator's `run` queue, and once for every process in the ready queue. These calls generate all the code for the transition matrix and for the case switches with the transition statements. The remaining procedures to make the analyzer run are mostly included in the file `pangen1.c` and are invoked in calls at the close of `gensrc()`.

The actual work of translating portions of the parse tree into C code happens in just two procedures: `putstmnt()` and `undostmnt()`. There is no magic here, just the generation of code, with some care taken to reduce the runtime requirements of validations. The state numbers are given by the `seqno` field in parse tree elements: every basic statement is assigned a unique sequence number by the parsing routines, as explained in Chapter 12. A set of transitions is assigned to every state to index the case switches. The transitions are numbered separately (note that there are likely to be more transitions than states if selection structures are used), and they are stored in the transition matrix.

Every sequence in a process body results in a call on procedure `putseq()`. The sequence is translated one statement at a time in a largely arbitrary order. Every element in the sequence that has been translated is labeled `DONE` in the status field. For the transitions the pointers between elements are followed, skipping as many intermediate steps as possible, using the routine `huntini()`. The actual code that reproduces the effect of a forward transition is generated by `putstmnt()`, listed in `pangen2.c`. The code that can undo the effect, when the depth first search unwinds, is generated by `undostmnt()`, listed in `pangen4.c`. Rather then giving a detailed expose of all the code being generated, let us consider the translation of one specific type of statement: an assignment.

The routine `putstmnt()` contains code which, after macro substitutions, amounts to the following:

```
case ASGN:      fprintf(fd, "(trpt+1)->oval = ");
                putstmnt(fd, now->lft,m,pid);
                fprintf(fd, ";\n\t\t");
                putstmnt(fd,now->lft,m,pid);
                fprintf(fd," = ");
                putstmnt(fd,now->rgt,m,pid);
                break;
```

Given the parse tree for the SPIN assignment

```
nips = 12+3*crunch;
```

this is translated into the sequence

```
case 34:        (trpt+1)->oval = now.nips;
                now.nips = (12 + (3 * ((P1 *)this)->crunch));
                m = 3; goto P333;
```

assuming that 34 is the number in the transition matrix assigned to the current transition, `nips` is a global variable, a permanent part of the state vector `now`, and `crunch` is a local variable that is accessible via the predefined pointer to the template of the current process in the state vector `this`. The first line is a backup of the old value of global `nips` in a special field of the stack that is used to organize the search in procedure `new_state()`. There is an offset of 1 to account for the fact that officially we do not know yet if the transition is going to be executable or not. Only if the execution is executable is the stack pointer increased, and the backup value will be in the right place for the undo operation.

The code for the generation of the matching undo operation looks as follows:

```
case ASGN:      putstmnt(tb, now->lft, m, pid);
                fprintf(tb, " = trpt->oval");
                checkchan(now->rgt, m, pid);
                break;
```

which for the same statement produces this code

```
case 28:        now.nips = trpt->oval;
                goto R333;
```

assuming again that 28 is the index assigned to the current undo operation in the transition matrix. The additional call on `checkchan()` in the undo code above is to make sure that no channels were created as a side effect of the assignment. If so, these channels are to be deleted again in the reverse transition.

13.7 GUIDED SIMULATION

The last extension to the simulator source code to be discussed is the implementation of procedure `match_trail()`. The code can be found in file `pangen5.c`. It looks for the simulation trail in file `pan.trail`, where the validator puts it. In its basic form, the trail has the following format:

```
0:0:1
1:0:2
2:0:3
3:0:4
4:0:5
5:0:6
6:0:8
7:3:15
```

Each line specifies a transition in three integer fields, separated by colons. The first field is a step number, counting up from zero to whatever the length of the error trail may be. The second field is the process number, with 0 for the `init` process, 1 for the first process that was started in a `run` statement, and so on. The last number on each line of the error trail identifies the state to which the process moves. The simulator's job is to follow the trail and touch upon all states listed. If, for now, we omit error recovery, temporal claim processes, and the treatment of stop states, the code looks as follows:

```
 1 match_trail()
 2 {    FILE *fd;
 3      int i, pno, nst;
 4
 5      if (!(fd = fopen("pan.trail", "r")))
 6      {       printf("spin -t: cannot find 'pan.trail'\n");
 7              exit(1);
 8      }
 9      Tval = 1; /* timeouts may be part of the trail */
10      while (fscanf(fd, "%d:%d:%d\n", &depth, &pno, &nst) == 3)
11      {       i = nproc - nstop;      /* number of running procs */
12              for (X = run; X; X = X->nxt)
13                      if (--i == pno) /* find process pno */
14                              break;
15              lineno = X->pc->n->nval;
16              do                      /* bring it to state nst */
17              {       X->pc = d_eval_sub(X->pc, pno, nst);
18              } while (X && X->pc && X->pc->seqno != nst);
19      }
20      printf("spin: trail ends after %d steps\n", depth);
21      wrapup();
22 }
```

After opening the trail file (lines 5-8), one directive at a time is read from the trail (lines 10). The right process is located (lines 12-14), and it is executed until the right state is reached (lines 16-18).

A stop state is identified by the new state 0, a non-existing state. It is executed as a removal of the process that was identified. The full code for `match_trail()` in Appendix E has extra checks to prepare it for cases where the validation model is unable to follow the trail, for instance if the model was changed since the trail was written. In these cases the simulator will report, for example,

```
step 23: lost trail (proc 4 state .13)
```

giving a rough indication where the simulation failed. In this case an inconsistency was discovered in step 23, just before process 4 reached state 13.

One case in which the simulator may lose track of the simulation trail, in a syntactically correct validation model, is illustrated by the following example.

```
do
:: (m >= N-1) -> break
:: (m <  N-1) -> m = m - 1
:: (m <  N-1) -> m = m - 1; n = n + 1
od
```

The two almost equal execution paths may, with the current implementation of the simulator, lead to an ambiguous trail. The problem can be avoided straightforwardly, by removing the ambiguity:

```
do
:: (m >= N-1) -> break
:: (m <  N-1) -> m = m - 1
        if
        :: skip
        :: n = n + 1
        fi
od
```

13.8 SOME APPLICATIONS

The analysis option is invoked from the original simulator with the flag -a, for instance, as follows:

```
$ spin -a factorial
```

At this point, typically within a second, we have generated a program that consists of five separate C files: a header file, the two case switches with forward and backward transitions, a main file with the main C routines, and a file with the transition matrix and some related routines.

```
$ wc pan.?
     55     197    1161 pan.b   # backward moves
    731    2159   14822 pan.c   # c routines
    108     409    2526 pan.h   # header
    120     482    2925 pan.m   # forward moves
    129     377    2580 pan.t   # transition matrix
   1143    3624   24014 total
```

The program can be compiled in two different ways. The default

```
$ cc -o pan pan.c
```

generates an analyzer that constructs a full state space, ruling out any chance of incompleteness. It provides 100% coverage, unless it runs out of memory. Optionally, a more frugal supertrace validator can be generated with the command

```
$ cc -DBITSTATE -o pan pan.c
```

In either case, the validation is started by typing

```
$ pan
pan: deadlock
pan: wrote pan.trail
...etc
```

If the validator finds an error it writes the simulation trail. The trail is used with the simulator in any of the modes discussed in Chapter 12, for instance with the `-s` option:

```
$ spin -t -s factorial
....etc.
```

where the new `-t` flag will tell the simulator to follow the trail in `pan.trail` rather than performing a random simulation.

Consider the following PROMELA version of Lynch's protocol, discussed in Chapters 2 and 5.

```
 1 #define MIN 9       /* first data message to send */
 2 #define MAX 12      /* last  data message to send */
 3 #define FILL        99      /* filler message */
 4
 5 mtype = { ack, nak, err }
 6
 7 proctype transfer(chan chin, chout)
 8 {   byte o, i, last_i=MIN;
 9
10     o = MIN+1;
11     do
12     :: chin?nak(i) ->
13             assert(i == last_i+1);
14             chout!ack(o)
15     :: chin?ack(i) ->
16             if
17             :: (o <  MAX) -> o = o+1        /* next */
18             :: (o >= MAX) -> o = FILL       /* done */
19             fi;
20             chout!ack(o)
21     :: chin?err(i) ->
22             chout!nak(o)
23     od
24 }
25
26 proctype channel(chan in, out)
27 {   byte md, mt;
28     do
29     :: in?mt,md ->
30             if
31             :: out!mt,md
32             :: out!err,0
33             fi
34     od
35 }
36
37 init
38 {   chan AtoB = [1] of { mtype, byte };
39     chan BtoC = [1] of { mtype, byte };
40     chan CtoA = [1] of { mtype, byte };
41     atomic {
42             run transfer(AtoB, BtoC);
43             run channel(BtoC, CtoA);
44             run transfer(CtoA, AtoB)
45     };
46     AtoB!err,0      /* start */
47 }
```

A few integer data messages are inserted into the system to allow us to look at at least a few message exchanges. We have also added a process type to model the expected behavior of the communication channel: randomly distorting messages. We can simulate the behavior of the system with the old simulator code, for instance

```
$ spin -s lynch
proc  0 (_init)         line  46, Send err,0    -> queue 1 (AtoB)
proc  1 (transfer)      line  22, Send nak,10   -> queue 2 (chout)
proc  2 (channel)       line  31, Send nak,10   -> queue 3 (out)
proc  3 (transfer)      line  14, Send ack,10   -> queue 1 (chout)
...etc.
```

This may or may not hit the assertion violation, depending on how the nondeterminism is resolved at each step.

For a validation of the same specification, we generate and compile the validation program, let's assume in supertrace mode, as follows:

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

We now have an executable program called `pan`. To see what options it accepts to perform the search, we can try

```
$ pan -?
unknown option
-cN stop at Nth error (default=1)
-l  find non-progress loops
-mN max depth N (default=10k)
-wN hash-table of 2^N entries (default=18)
```

We can, for instance, set the maximum search depth (the size of the backtrace stack) to another value than the default of 10,000 steps, or we can change the size of the hash table.

- With full state space storage, the size of the hash table should be chosen larger than or equal too the total number of reachable states that is expected, to avoid a serious time penalty for the resolution of the hash collisions (see Chapter 11).
- In supertrace mode the size of the hash-table is equal to the number of bits in the state space, so the `-w` flag really selects the actual size of the state space that is used for the search. By default this state space is set to 2^22 = 4,194,304 bits = 524,288 bytes. The size of the state space determines the maximum number of states that can be analyzed. For the default case this is roughly 4,194,304/2 = 2,097,152 states, independent of the size of the state vector. (In this implementation two bits are used for every state stored.)

The coverage of the search will be smaller as we get closer to that limit. We discuss an indicator of that coverage, the hash factor, with a few other examples later. We first try the validator in exhaustive mode.

```
$ pan
assertion violated (i==(last_i+1))
pan: aborted (at depth 53)
pan: wrote pan.trail
full state space search for:
        assertion violations and invalid endstates
search was not completed
vector 56 byte, depth reached 53, errors: 1
      58 states, stored
       2 states, linked
       1 states, matched           total:       61
hash conflicts: 0 (resolved)
(size 2^18 states, stackframes: 0/16)
```

At the end of each run the validator prints the numbers of states stored, linked and matched. Stored states are states that have been added to the state space, either in full or in compressed form as two bits, depending on how the program was compiled. Linked states are states that were encountered within an atomic sequence, no state space checks are performed on them. Matched states are states that were analyzed and later revisited.

The validator found an error that is documented in the file `pan.trail`. We can now feed back this error trail to the simulator to look precisely at what goes on. For instance:

```
$ spin -t -s -r lynch
proc  0 (_init)         line  46, Send err,0    -> queue 1 (AtoB)
proc  1 (transfer)      line  21, Recv err,0    <- queue 1 (chin)
proc  1 (transfer)      line  22, Send nak,10   -> queue 2 (chout)
proc  2 (channel)       line  29, Recv nak,10   <- queue 2 (in)
proc  2 (channel)       line  32, Send err,0    -> queue 3 (out)
proc  3 (transfer)      line  21, Recv err,0    <- queue 3 (chin)
proc  3 (transfer)      line  22, Send nak,10   -> queue 1 (chout)
proc  1 (transfer)      line  12, Recv nak,10   <- queue 1 (chin)
proc  1 (transfer)      line  14, Send ack,10   -> queue 2 (chout)
....
proc  3 (transfer)      line  21, Recv err,0    <- queue 3 (chin)
proc  3 (transfer)      line  22, Send nak,99   -> queue 1 (chout)
proc  1 (transfer)      line  12, Recv nak,99   <- queue 1 (chin)
spin: "lynch" line 13: assertion violated
#processes: 4
                _p = 3
proc  3 (transfer)      line 11 (state 15)
proc  2 (channel)       line 28 (state 6)
proc  1 (transfer)      line 13 (state 3)
proc  0 (_init) line 47 (state 6)
4 processes created
```

The simulator run can be repeated with different flags, e.g., printing variable values and process states, until the cause of the error is determined and can be repaired.

13.9 COVERAGE IN SUPERTRACE MODE

The coverage of the search in supertrace mode is determined by the number of hash collisions that occur. This number, of course, is usually unknown. It can be determined by comparing a run with full state storage to a run with a bit state space, but this is not always feasible. The number of hash collisions, however, depends critically on the ratio of the size of the hash-table, i.e., the number of bits in the state space, and the number of states that is stored. We call this factor the *hash factor*. It is calculated by the validator after each run as the size of the hash-table divided by the number of states stored. A high number (more than 100) correlates with good coverage. Low numbers (near 1) imply poor coverage.

Since we store two bits per state in supertrace mode, the hash factor can be anywhere from `2^N` up to and including 0.5, where `N` can be set by the user to grab the maximum amount of memory that is available on the target machine. (For full state space storage the lower limit on the hash factor is zero.) By empirical testing with full and bit state space runs it can be confirmed that a hash-factor of 100 or more virtually guarantees a coverage of 99% to 100% of all reachable states. As an example, Table 13.1 gives the results of tests with a protocol model that has 334,151 reachable states.

The original run in this case was the full state space version, using 45.6 Mbyte of memory. It stored all states and resolved a total of 66,455 hash conflicts on the way. The run was repeated, first with a supertrace validation using the flag `-w25`, giving a hash factor of 100.9 and a coverage of 99.45%. By virtue of the double bit hash function, the number of hash conflicts is substantially lower than in the first run. The precise number can be found by subtracting the number of reached and stored states in the first run from the number of reached (but not stored) states in the second run. In each of the next three supertrace validations we halved the hash factor by using the flags `-w24`, `-w23`, and `-w22`. The last run uses no more than 6.3 Mbyte of memory, of which 6 Mbyte, in both the full state space storage version and the supertrace version, is used for storing the backup trail which was 300,000 steps long for all runs of this test protocol. (This also explains why the amount of memory required does not precisely half each time the argument to the `-w` flag is decremented.)

Table 13.1 – Correlation between Hash Factor and Coverage

Search	Hash Factor	States Stored	Hash Collisions	Memory Used	Coverage
exhaustive	—	334,151	66,455	45.6 Mb	100%
supertrace	100.9	332,316	1,835	9.9 Mb	99.45%
supertrace	50.9	329,570	4,581	7.9 Mb	98.62%
supertrace	25.7	326,310	7,841	6.9 Mb	97.65%
supertrace	13.0	322,491	11,660	6.3 Mb	96.51%

For comparison, a run of the full state space storage method that is restricted to 6.3 Mbyte of memory to store its state space, predictably, gets less coverage. The exhaustive search effectively degrades into an uncontrolled partial search, as illustrated in Table 13.2.

Table 13.2 – Coverage of Partial Searches

Search	Hash Factor	States Stored	Hash Collisions	Memory Used	Coverage
exhaustive	—	83,961	389,671	6.3 Mb	25.12%
supertrace	13.0	322,491	11,660	6.3 Mb	96.51%

The bit state space is clearly the method of choice here.

13.10 SUMMARY

Many automated validation tools require considerable effort from the user to translate a validation model into the low level code that is used to run the validator. The interpretation of error reports produced by those tools similarly can require considerable human ingenuity. With the simulator and validator generator SPIN, and the validation language PROMELA, we have tried to provide a high level design environment in which everything from simple protocols, up to complete designs for distributed message passing systems can be thoroughly tested and debugged before they are implemented. These tools can help us to deal effectively with the notoriously difficult problems of asynchrony and concurrency. The tools are portable, powerful, and efficient.

EXERCISES

13-1. Consider how the code must be changed to replace the depth-first search order with breadth-first. What are the memory requirements? □

13-2. Modify the code to optimize the implementation of send and receive routines, and measure its effect. □

13-3. Add an option to SPIN for restricting the validation runs to ''fair'' executions. This option is based on the assumption of a ''fair process scheduler.'' This means that any process that can execute a statement is assumed to be enabled to do so within finite time. All infinite executions (cycles) that violate this fairness assumption can be ignored. All non-progress loops or acceptance cycles that violate this assumption should similarly be ignored. Hint: perform an extra check before reporting any error in cyclic sequences. □

13-4. (E.A. Emerson - P. van Eijk) Implement a method that can give a better prediction of the coverage of partial supertrace validations. Do this by starting a supertrace validation by selecting 1000 states at random from the state space. (How?) Store those full states in a separate lookup table and check during the supertrace validation how many of those 1000 states are reached. The fraction of the states reached is an indication of the coverage. How reliable is the estimate? How expensive? □

13-5. Modify the validator generator to allow for the automatic generation of protocol implementations from PROMELA code. Note that C code is already generated for all transitions and actions. Replace the search procedure `new_state()` with a scheduler, as used in the simulator code (Chapter 12), and allow for certain channels to be identified as special device channels (for example, files) that can be linked to C library routines that access the raw I/O channels. Your solution need not contain more than two pages of code. □

BIBLIOGRAPHIC NOTES

The validator described has several predecessors of varying scope and performance. For those interested, the papers Holzmann [1984a, 1985, 1988] document the more significant changes. The last of these papers contains a detailed explanation of the state vector model and the bit state space method. The method described in Holzmann [1988] is the only version from this sequence that achieves a better performance, in terms of runtime and memory usage, than the method described here. It requires substantially more code to implement.