Chapter 15. Sample SPIN Models

> "Few things are harder to put up with than the annoyance of a good example."
>
> —(Mark Twain, 1835–1910)

In this chapter we will discuss a few small PROMELA models that exploit some interesting and possibly useful features of the specification language. We will focus mostly on language and modeling issues here. More examples of PROMELA models can be found in the standard SPIN distribution.

Eratosthenes

Our first example is a PROMELA version of an ancient algorithm for finding primes by counting off numbers and systematically intercepting the non−primes among them. The algorithm, a favorite programming exercise today, is due to the Greek philosopher and mathematician Eratosthenes of Cyrene (a city in modern day Libya which is today called Shahhat). Figure 15.1 shows a version in PROMELA that makes use of the fact that we can use dynamic process creation and pass channel identifiers between running processes.

Figure 15.1 The Sieve of Eratosthenes

```
 1 /*
 2     The Sieve of Eratosthenes (c. 276-196 BC)
 3     Prints all prime numbers up to MAX
 4 */
 5 #define MAX 25
 6
 7 mtype = { number, eof };
 8
 9 chan root = [0] of { mtype, int };
10
11 proctype sieve(chan c; int prime)
12 {    chan child = [0] of { mtype, int };
13      bool haschild;
14      int n;
15
16      printf("MSC: %d is prime\n", prime);
17 end: do
18      :: c?number(n) ->
19         if
20         :: (n%prime) == 0 ->
21            printf("MSC: %d = %d*%d\n", n, prime, n/prime)
22         :: else ->
23         if
24            :: !haschild -> /* new prime */
25                   haschild = true;
26                   runsieve(child, n);
27            :: else ->
28                   child!number(n)
29            fi;
30         fi
31      :: c?eof(0) ->
32         break
33      od;
34      if
35      :: haschild ->
36         child!eof(0)
37      :: else
38      fi
39 }
40
41 init
42 {   int n = 2;
43
44     run sieve(root, n);
45     do
46     :: (n < MAX) -> n++; root!number(n)
47     :: (n >= MAX) -> root!eof(0); break
```

```
48      od
49 }
```

Because a PROMELA model must always be finite, we have to place an upper−bound on the largest integer value that we will test for primality. SPIN is not designed to handle computational problems, so do not expect to get away with a very large bound here. The bound is defined in Figure 15.1 in a macro definition named `MAX`. We have used the value 25. Only two types of messages are used, defined in an `mtype` declaration, and named `number` and `eof`. The latter type of message is used to trigger an orderly termination of the system of processes when the test for primality of the number with the maximal value allowed has been completed.

Our system of processes starts off with just a single running process: `init`. The principle of operation of the algorithm is that we test integer numbers one by one, in ascending order. We will start off assuming that we know only that two is a prime. Clearly, for any higher number to be prime, it should minimally not be divisible by two, so the first thing that the initial process will do is to start up a tester for that value. The initial process does so by creating a first process of type `sieve`, and passing it in an argument the value two as a first prime number to use in the tests. Also passed is an argument is the name of the channel that the initial process will use to communicate further information to the sieve process. For the first process this is a globally declared rendezvous channel named `root`.

Once the first test process is set up, the initial process will simply pass all integer numbers greater than two, up to the preset maximum, to its newly created child. It is the child's job now to figure out if the numbers passed to it are prime, and it is free to create its own children to help do the job.

When a process of type `sieve` starts up, the first thing it will do is to acknowledge the fact that it was passed, what it trusts is, a prime number as an argument. It does so by printing the number (line 16 in Figure 15.1), using the prefix `MSC:` to make sure that this line of output will be picked up in the message sequence charts that can be created by XSPIN. Next, it stops and waits for input to arrive on the channel that was passed to it by its parent.

One of only two types of messages can arrive, as shown on line 18 and line 31 in Figure 15.1.

A message of type `number` carries an integer number that is to be tested for primality. Every single instantiation of the sieve process will test if the number is divisible by the prime number it was passed by its parent. If divisible, the number is not prime, and that fact is printed. Otherwise, the number is passed to the next process to test for possibly more known primes. If no next process exists yet, the value of local boolean variable `haschild` will still be false (the default initial value). The sieve process will now clone itself and start up a new copy of `sieve`, passing the newly discovered prime number as an argument, as well as the name of the local channel `child` that it will use to pass new numbers.

If a child process already exists, that means that more tests for primality have yet to be done before this new number can be declared prime. The number is simply sent to the child process over the local channel, and the test process is repeated.

Meanwhile, the initial process can be sending a new number into the pipeline of primality testers, and in principle all processes can be active simultaneously, each testing the divisibility of a different number against the prime number they each hold. A simulation run might proceed as follows:

```
$ spin eratosthenes
      MSC: 2 is prime
          MSC: 3 is prime
```

```
      MSC: 4 = 2*2
              MSC: 5 is prime
      MSC: 6 = 2*3
      MSC: 8 = 2*4
          MSC: 9 = 3*3
                  MSC: 7 is prime
      MSC: 10 = 2*5
      MSC: 12 = 2*6
      MSC: 14 = 2*7
                       MSC: 11 is prime
      MSC: 16 = 2*8
          MSC: 15 = 3*5
                            MSC: 13 is prime
      MSC: 18 = 2*9
      MSC: 20 = 2*10
          MSC: 21 = 3*7
                                 MSC: 17 is prime
      MSC: 22 = 2*11
      MSC: 24 = 2*12
              MSC: 25 = 5*5
                                       MSC: 19 is prime
                                           MSC: 23 is prime
10 processes created
```

Although the algorithm itself is deterministic, the process scheduling is not, and in different runs this can cause print statements to appear in slightly different orders. Ten processes were created, one of which is the initial process. This means that the algorithm accurately found the nine prime numbers between one and 25. When the maximal number is reached, the `eof` messages is passed down the chain all the way from the initial process to the most recently created sieve process, and all processes will make an orderly exit.

A verification run with the model as specified is uneventful:

```
(Spin Version 4.0.7 -- 1 August 2003)

        + Partial Order Reduction

Full statespace search for:
        never claim   - (none specified)
        assertion violations +
        acceptance cycles  - (not selected)
        invalid end states  +

State-vector 284 byte, depth reached 288, errors: 0
    2093 states, stored
     478 states, matched
    2571 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 1 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
0.611 equivalent memory usage for states ...
0.508 actual memory usage for states (compression: 83.13%)
      State-vector as stored = 235 byte + 8 byte overhead
1.049 memory used for hash table (-w18)
0.240 memory used for DFS stack (-m10000)
```

```
1.698 total actual memory usage

unreached in proctype sieve
        (0 of 25 states)
unreached in proctype :init:
        (0 of 11 states)
```

There are no deadlocks and there is no unreachable code, as we would expect. The partial order reduction algorithm could in principle work better, though, if we can provide some extra information about the way that the initial and the sieve processes access the message channels. In principle, this is not too hard in this case. On line 15, for instance, we can try to add the channel assertions

```
15 xr c; xs child;
```

because the sieve process is guaranteed to be the only process to read from the channel that was passed to it as an argument, and the only one to send messages to the channel it will use to communicate with a possible child process. Similarly, the assertion

```
43 xs root;
```

could be included on line 43 in the `init` process to assert that the initial process is the only process to send messages to channel `root`. If we do so, however, the verifier will warn us sternly that channel assertions are not allowed on rendezvous channels.

```
$ spin -a eratosthenes
$ cc -o pan pan.c
$ ./pan
chan root (0), sndr proc :init: (0)
pan: xs chans cannot be used for rv (at depth 0)
pan: wrote eratosthenes.trail
...
```

We can correct this by turning the two rendezvous channels declared on lines 9 and 12 in Figure 15.1 into buffered message channels with the minimum storage capacity of one message. Line 9 in Figure 15.1 then becomes:

```
9 chan root = [1] of { mtype, int };
```

Similarly, line 12 is now written:

```
12 { chan child = [1] of { mtype, int };
```

This of course in itself will increase the number of potentially reachable states, since it decouples the process executions a little more. Repeating the verification confirms this. If the channel assertions are not included, the number of reachable states now increases tenfold (to 24,548). With the channel assertions, however, the size of the state space decreases tenfold to a mere 289 reachable states, which provides a compelling illustration of the effectiveness of channel assertions.

In this first model we are using one process for each prime number that is found. Because there cannot be more than 255 running processes in a SPIN model, we cannot use this model to find more than only the first 254 prime numbers greater than one. This means that a value for `MAX` greater than 1,609 (the 254th prime) would be of little use, unless we can somehow rearrange the code to avoid the dynamic creation of processes. This is not too hard, as shown in Figure 15.2, though the resulting model is not quite as elegant.

Figure 15.2 Alternative Structure for Sieve

```
 1 mtype = { number, eof };
 2
 3 chan found = [MAX] of { int };
 4
 5 active proctype sieve()
 6 {  int n = 3;
 7    int prime = 2;
 8    int i;
 9
10    found!prime;
11    printf("MSC: %d is prime\n", prime);
12    do
13    :: n < MAX ->
14       i = len(found);
15       assert(i > 0);
16       do
17       :: i > 0 ->
18           found?prime;
19           found!prime; /* put back at end */
20           if
21           :: (n%prime) == 0 ->
22           /* printf("MSC: %d = %d*%d\n",
23                       n, prime, n/prime); */
24              break
25           :: else ->
26               i--
27           fi
28       :: else ->
29           break
30       od;
31       if
32       :: i == 0 ->
33           found!n;
34           printf("MSC: %d is prime number %d\n",
35                        n, len(found))
36       :: else
37       fi;
38       n++
39    :: else ->
40       break
41    od
```

```
42 }
```

This time we store prime numbers in a channel, and retrieve them from there for primality testing. We have set the capacity of the channel generously to the value of MAX, although a much smaller value would also suffice. Only a number that is not divisible by any of the previously discovered primes is itself prime and can then be added into the channel. In this version of the sieve process we have left the macro MAX undefined, which means that we can now pass a value in via a command–line argument to SPIN. We can now surpass the old limit of 254 primes easily, for instance, as follows:

```
$ spin -DMAX=10000 eratosthenes2
      MSC: 2 is prime
      MSC: 3 is prime nr 2
      ...
      MSC: 9941 is prime nr 1226
      MSC: 9949 is prime nr 1227
      MSC: 9967 is prime nr 1228
      MSC: 9973 is prime nr 1229
1 process created
```

If we repeat the verification attempt for the alternative model, using the same value for MAX as before, we see that the number of states has increased a little compared to the best attempt from before using channel assertions.

```
$ spin -DMAX=25 -a eratosthenes2
$ cc -o pan pan.c
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
       + Partial Order Reduction

Full statespace search for:
       never claim          - (none specified)
       assertion violations +
       acceptance cycles    - (not selected)
       invalid end states   +

State-vector 132 byte, depth reached 479, errors: 0
     480 states, stored
       0 states, matched
     480 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493 memory usage (Mbyte)

unreached in proctype sieve

        (0 of 32 states)
```

However, we can also note that the size of the state vector has decreased from 284 bytes in the first model, which increases with MAX, to a fixed size of just 132 bytes in the new model. This means that the 289 states

from before will actually take up more memory than the 480 states from the new model. The simpler model usually wins in a battle for complexity control.

Process Scheduling

The next problem concerns the design of a reasonably efficient method for scheduling process execution in a multiprocessor system. The processes compete for access to shared resources, and they may have to be suspended when a resource is temporarily unavailable. The process suspension is done with a system call named `sleep`, which also records the particular resource that the process is waiting to access. When a process releases a resource, it calls the routine `wakeup`, which checks if any processes are currently suspended, waiting for the resource being released, and if so resumes execution of those processes. The data structures that record the state of the resource, and the data structures that record the state of the processes, are themselves also shared resources in the system, and access to them has to be protected with locks. In a uniprocessor system simply masking interrupts can suffice to lock out competing processes while operations on shared data structures are performed, but in a multiprocessor system this is not sufficient and we need to rely on higher–level locks.

In most systems, the availability of a global, indivisible test and set instruction can be assumed to solve this problem. If, for instance, we have a lock variable named `lk`, the indivisible test and set instruction, which is called `spinlock` in the UTS system, can be modeled in PROMELA as

```
#define spinlock(lk) atomic { (lk == 0) -> lk = 1 }
```

and the matching lock release operation as

```
#define freelock(lk) lk = 0
```

The scheduling problem is easy to solve if we would allow a process to simply set the `spinlock` for the duration of all access to the resource: it would effectively lock out all other processes. Such a solution would be very inefficient, though, forcing other processes to continue executing while competing to set the lock variable. The real challenge is to minimize the use of global locks, suspending process executions where possible, while securing that no process can accidentally be suspended forever. The latter problem is called a "missed wakeup."

The algorithm that was adopted for the Plan9 operating system was discussed in Pike et al. [1991], including a verification with and early version of SPIN. Another solution was proposed in Ruane [1990] for use in Amdahl's UNIX time sharing system, UTS®. We will consider Ruane's method here. An earlier discussion of this method appeared in Holzmann [1997b] with a commentary, exposing some flaws in that discussion, appearing in Bang [2001].

For our current purpose it is sufficient to restrict the number of shared resources in the system to just one single resource. This resource can be represented in the C implementation by a data structure of the following type:

```
typedef struct R {
```

```
        int lock;  /* locks access to resource */
        int wanted; /* processes waiting */
        ...   /* other fields */
} R;

R *r;   /* pointer to resource structure */
```

A process that gains access to the resource will set the `lock` field in the resource data structure to record that the resource is in use. If a process finds the resource locked, it suspends itself after setting the `wanted` flag to one, to record that at least one process is waiting for the resource to be released. A process that releases the resource first checks the `wanted` flag to see if any processes are waiting, and if so it will restart those processes one by one. Each such process then retests the resource `lock` field to try again to gain access to the resource.

In the UTS solution, a process can be suspended while holding the global lock variable with a variant of the `sleep` routine, called `sleepl`. The implementation of `sleepl` then has to make sure that the global lock is released while the process is suspended and reacquired when it is resumed.

To acquire access to the resource, the code that was used would first set the spinlock on `lk`, and then test the value of the `lock` variable from the resource, as follows:

```
spinlock(&lk);
while (r->lock) {
       r->wanted = 1;
       sleepl(r, &lk);
}
r->lock = 1;
freelock(&lk);
```

As a minor detail, note that in the C code a pointer to lock variable `lk` is passed to the routines `spinlock` and `freelock`, where in the PROMELA version we passed the variable name itself.

To release the resource, a process executes the following piece of code:

```
r->lock = 0;
waitlock(&lk);
if (r->wanted) {
         r->wanted = 0;
         wakeup(r);
}
```

If the `wanted` flag indicates that at least one process is waiting to access the resource, the waiting processes are restarted through a call to the `wakeup` routine. A `waitlock` is used here instead of a `spinlock`. The `waitlock` primitive can be modeled in PROMELA as follows:

```
#define waitlock(lk) (lk == 0)
```

Ruane reported that some time after these routines had been implemented a race condition was discovered that could lead to a process being suspended without ever being resumed. After analyzing the problem, the designers of the code proposed a change in the wakeup routine that looked as follows:

```
r->lock = 0;
waitlock(&lk);
if (r->wanted) {
         r->wanted = 0;
         waitlock(&lk);
         wakeup(r);
}
```

For a while, no further problems were detected, but the designers had a lingering doubt about the adequacy of their fix. They asked two specific questions:

- Could a SPIN verification have found the problem?
- Is the modified version free from other race conditions?

To answer these questions, we will build a basic SPIN model without resort to embedded C code. This means that we cannot use pointers, and we will have to make some changes in the way that the resource flags are specified. All changes are relatively minor.

We begin by modeling the effect of the system routines `sleepl` and `wakeup` with two `inline` definitions. To do so, we have to decide how to represent the process states. We use a one–dimensional array to record process states, indexed by the process instantiation numbers, as follows:

```
mtype = { Wakeme, Running };   /* process states */
mtype pstate[N] = Running;
```

The initial state of each process is `Running`. Access to the process states has to be protected by a special lock, which we call `sq`. The `wakeup` routine, shown in Figure 15.3, acquires this lock and checks if any processes are suspended on access to the resource. If it finds any, it moves them back into the `Running` state.

Figure 15.3 Sleep–Wakeup Routines

```
1 #define N 3                     /* nr of processes */
2
3 mtype = { Wakeme, Running };    /* process states  */
4
5 mtype pstate[N] = Running;
6
7 bit r_lock;
8 bit r_wanted;                  /* resource state */
9 bit lk, sq;                    /* locks */
10
11 #define freelock(x) x = 0
12 #define waitlock(x) (x == 0)
13 #define spinlock(x) atomic { waitlock(x) -> x = 1 }
14
```

```
15 inline wakeup(x) {
16     spinlock(sq);
17     i = 0;
18     do :: i < N ->
19             if
20             :: pstate[i] == Wakeme ->
21                     pstate[i] = Running
22             :: else -> i++
23             fi
24         :: else -> break
25     od;
26     freelock(sq)
27 }
28 inline sleepl(y, x) {
29     spinlock(sq);
30     freelock(x);
31     pstate[_pid] = Wakeme;
32     freelock(sq);
33     (pstate[_pid] == Running);
34     spinlock(x)
35 }
36
```

The `sleepl` routine changes the process state, again under protection of the `sq` lock, and it releases the global lock. The lock is reacquired when the process is moved back into the `Running` state. The first argument to `sleepl`, which points to the resource data structure in the original code, can be ignored here since we consider access to only a single resource.

In the verification model from Figure 15.4, a user process alternately tries to gain access to the resource and then release it, following the proposed UTS code for the calls on `sleepl` and `wakeup`.

Figure 15.4 Remainder of Verification Model for UTS

```
37 active [N] proctype user()
38 { pid i;
39   do :: spinlock(lk);        /* spinlock(&lk);    */
40      do :: r_lock ->         /* while (r->lock) { */
41            r_wanted = 1;     /* r->wanted = 1;    */
42            sleepl(_, lk)     /* sleepl(r, &lk);   */
43         :: else -> break
44      od;                     /* }                 */
45      r_lock = 1;             /* r->lock = 1;      */
46      freelock(lk);           /* freelock(&lk);    */
47
48 R:   /* use resource r */
49
50      r_lock = 0;            /* r->lock = 0;       */
51      waitlock(lk);          /* waitlock(&lk);     */
52      if
53      :: r_wanted ->      /* if (r->wanted) {      */
54         r_wanted = 0;       /* r->wanted = 0;     */
55 #ifdef FIX
56        waitlock(lk);        /* waitlock(&lk);     */
57 #endif
58        wakeup(_);           /* wakeup(r);         */
59     :: else
60     fi                      /* }                  */
61  od
62 }
```

The proposed fix can be included into, or excluded from, the model by defining or undefining the preprocessor directive named `FIX`. The original C code is placed next to the PROMELA code in comments to show the correspondence. Apart from syntax, there is a fairly close match.

To verify the model we must now formulate a correctness property. To show that there can be no missed wakeups, we should be able to show, for instance, that it is impossible for any process to remain in the `Wakeme` state forever. If we define the propositional symbol `p` as:

```
#define p (pstate[0] == Wakeme)
```

it should be impossible for `p` to remain true infinitely long in an execution of the system. The corresponding LTL formula is `<>[]p`. The claim that can now be appended to the model is generated with SPIN as follows:

```
$ spin -f '<>[]p'
never { /* <>[]p */
T0_init:
        if
        :: ((p)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: ((p)) -> goto accept_S4
        fi;
}
```

The process instantiation numbers are 0, 1, and 2 in this model. Because the system is symmetrical, it should not matter which process we select for the check in property `p`.

The verifier is now first generated and compiled without enabling the fix, as follows:

```
$ spin -a uts_model   # no FIX
$ cc -o pan pan.c
```

The verifier quickly finds a counterexample. We can produce a relatively

short error trail by restricting the search depth to 60 steps.

```
$ ./pan -a -m60
...
pan: acceptance cycle (at depth 42)
pan: wrote uts_model.trail
...
```

The error sequence can be reproduced with SPIN's guided simulation option, for instance, as in:

```
$ spin -t -p uts_model
...
```

For clarity, we will edit the output a little here to indicate the sequence of steps taken by each of the three processes, using macro and inline names from the model to shorten the trail some more. We list the actions of the user processes in three columns, in the order of their process instantiation numbers. The first step is taken by the user process with pid number 2, which appears in the third column.

```
                /* pid 2 */
1               spinlock(lk)
2               else
3               r_lock = 1
4               freelock(lk)
5               /* use resource */
6               r_lock = 0
      /* pid 1 */
7     spinlock(lk)
8     else
9     rlock = 1
10    freelock(lk)
11    /* use resource */
            /* pid 2 */
12          waitlock(lk)
   /* pid 0 */
13 spinlock(lk)
14 (r_lock)
15 r_wanted = 1
                 /* pid 2 */
16               (r_wanted)
17               r_wanted = 0
   /* pid 0 */
18 sleepl(r,lk)
                 /* pid 2 */
19               waitlock(lk)
20                wakeup(r)
      /* pid 1 */
21    <<<CYCLE>>>
22    r_lock = 0
23    waitlock(lk)
24    else
25    spinlock(lk)
26    else
27    r_lock = 1
28    freelock(lk)
29    /* use resource */
```

In this scenario, the user process with pid 0, executing at steps 13, 14, 15, and 18, is indeed indefinitely held in its `Wakeme` state, but the scenario also shows that the processes with pid 2 is assumed to be delayed indefinitely in its call of the `wakeup` routine, trying to acquire the spinlock inside this call in step 20.

The spinlock on `lk` is repeatedly set and released by the remaining process in steps 25 and 28.

This is a valid but not an interesting counterexample because it assumes unfair process scheduling decisions. To home in on the more interesting cases, we have to add fairness constraints to the property. Our verification model already contains the label `R` at the point where access to the resource is obtained. We can extend the property to state that it should be impossible for one of the processes to remain in its `Wakeme` state, only while the other two processes continue to access the resource. The system is symmetrical, so it should not matter which process we pick for the check. The new property can be expressed in LTL formula

```
<>[]p && []<>q && []<>r
```

with the propositional symbol definitions:

```
#define p  (pstate[0] == Wakeme)
#define q  (user[1]@R)
#define r  (user[2]@R)
```

If we repeat the verification, again for the model without the fix enabled, we can obtain another error scenario, now slightly longer. The essence of this scenario can be summarized in the following steps:

```
1       spinlock(lk)
2       else
3       r_lock = 1
4       /* use resource */
5       freelock(lk)
6       r_lock = 0
7       waitlock(lk)

8               spinlock(lk)
9               else
10              r_lock = 1
11              freelock(lk)
12              /* use resource */

13 spinlock(lk)
14 (r_lock)

15              r_lock = 0

16 r_wanted = 1

17     (r_wanted)
18     r_wanted = 0
19     wakeup(_)

20 sleepl(_,lk)
```

The process with pid 1 accesses the resource and releases the resource lock in step 6. It is about to check, in step 17, if any processes are suspended, waiting to access the resource. Meanwhile, the process with pid 2 acquires access in steps 8–12, and causes the process with pid 1 to prepare for a call on `sleepl`, after finding the lock set, in step 14. The process sets the `wanted` flag, which is immediately detected and cleared by the process with pid 1. This process now proceeds with the execution of the `wakeup` routine, but process 0 has not actually been suspended yet. As a result, when process 0 finally suspends itself, the `wanted` flag is zero, which means that it can no longer be detected.

The process can now remain suspended indefinitely, while the other processes continue to acquire and release the resource.

We can now repeat the verification with the proposed fix enabled. The relevant part of the output is as follows:

```
$ spin -a -DFIX uts_model       # include the FIX
$ cc -o pan pan.c
$ ./pan -a
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction
Full statespace search for:
        never claim    +
        assertion violations    + (if within scope of claim)
        acceptance cycles       + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 44 byte, depth reached 5619, errors: 0
   47835 states, stored (49399 visited)
   56388 states, matched
  105787 transitions (= visited+matched)
       0 atomic steps
```

The state space for this three–process model is still relatively small, with under 50,000 reachable states. The verification run shows that the fix does indeed secure that the correctness property can no longer be violated.

Out of curiosity, we can also repeat the last run, but this time leave out the LTL property, to see how much complexity the verification of the claim added above the complexity of a basic reachability analysis for safety properties. We proceed as follows.

```
$ cc -DNOCLAIM -o pan pan.c
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        acceptance cycles       - (not selected)
        invalid end states      +

State-vector 40 byte, depth reached 2809, errors: 0
   42983 states, stored
```

```
43494 states, matched

86477 transitions (= stored+matched)
0      atomic steps
```

Note that the inclusion of the LTL property increased the state space by just 4,852 reachable states, or just under 11.3%. Note also that the use of the nested depth−first search algorithm causes the depth reached in the state space to double in this case.

A Client–Server Model

It is relatively simple to create SPIN models with a dynamically changing number of active processes. Each newly created process can declare and instantiate its own set of local variables, so through the creation of a new process we can also create additional message channels. It may be somewhat confusing at first that message channel identifiers can have a process local scope, if declared within a `proctype` body, but that the message channels themselves are always global objects. The decision to define channels in this way makes it possible to restrict the access to a message channel to only specifically identified processes: message channels can be passed from one process to another. We will use this feature in the design of a simple, and fairly generic client–server model.

We will design a system with a single, fixed server that can receive requests from clients over a known global channel. When the server accepts a request for service, it assigns that request to an agent and provides a private channel name to the client that the client can use to communicate with the agent. The remainder of the transaction can now place between agent and client, communicating across a private channel without further requiring the intermediacy of the server process. Once the transaction is complete, the agent returns the identifier for the private channel to the server and exits.

Figure 15.5 shows the design of the agent and server processes. The fixed global channel on which the server process listens is declared as a rendezvous port called `server`. The server process has a private, locally declared, set of instantiated channels in reserve. We have given the server process a separate local channel, named `pool`, in which it can queue the channels that have not yet been assigned to an agent. The first few lines in the server process declaration fill up this queue with all available channels.

Figure 15.5 Agent and Server Processes

```
#define N          2

mtype = { request, deny, hold, grant, return };

chan server = [0] of { mtype, chan };

proctype Agent(chan listen, talk)
{
        do
        :: talk!hold(listen)
        :: talk!deny(listen) -> break
        :: talk!grant(listen) ->
wait:           listen?return; break
        od;
        server!return(listen)
}

active proctype Server()
{       chan agents[N] = [0] of { mtype };
        chan pool = [N] of { chan };
        chan client, agent;
        byte i;

        do
        :: i < N -> pool!agents[i]; i++
        :: else -> break
        od;
```

```
end:    do
        :: server?request(client) ->
                if
                :: empty(pool) ->
                        client!deny(0)
                :: nempty(pool) ->
                        pool?agent;
                run Agent(agent,client)
                fi
        :: server?return(agent) ->
                pool!agent
  od
}
```

A client sending a request to the server attaches the name of the channel where it will listen for responses from the server or the server's agent. If the channel pool is empty at this point, the server has no choice but to deny the request immediately. If a channel is available, an agent process is started and the name of the new private channel is passed to that agent, together with the channel through which the client can be reached. The server now goes back to its main loop, waiting for new client requests. Eventually, when the client transaction is complete, the server's agent will return the now freed up private channel, so that the server can add it back into its pool of free channels.

We have set up the agent to randomly decide to either grant or deny a request, or to inform the client that the request is on hold. (Think of a library system, where a user can request books. In some cases a book can be on loan, and the user may be informed that the book was placed on hold.) If the request is granted, the agent will move to a wait state where it expects the client to eventually send a return response, signifying that the transaction is now complete. The agent process will now notify the server that the private channel can be freed up again, and it terminates.

Figure 15.6 shows the structure of the client process for this system. The client has its own private channel that it reserves for communications with the server. It initiates the communication, after a `timeout` in this case, by sending a first request to the server on the known global channel, with its own channel identifier attached. It will now wait for the response from either the server or the server's agent. A denial from the server brings the client back to its initial state, where it can repeat the attempt to get a request granted. A hold message is simply ignored by this client, although in an extended model we could consider giving the client the option of canceling its request in this case. When the request is granted, the client will faithfully respond with a return message, to allow the server's agent to conclude the transaction.

Figure 15.6 The Client Processes

```
#define M        2

active [M] proctype Client()
{       chan me = [0] of { mtype, chan };
        chan agent;

end:    do
        :: timeout ->
               server!request(me);
               do
               :: me?hold(agent)
               :: me?deny(agent) ->
                       break
               :: me?grant(agent) ->
                       agent!return;
```

```
                    break
                od
        od
}
```

In this model, we have both dynamic process creation and the passing of channel identifiers from one process to the next. Dynamic process creation in a model such as this one can sometimes hold some surprises, so it will be worth our while to try some basic verification runs with this model. Clearly, the complexity of the model will depend on the number of client processes and the maximum number of agents that the server can employ. We will start simple, with just two client processes and maximally two agents. The verifi−cation then proceeds as follows:

```
$ spin -a client_server.pml
$ cc -o pan pan.c
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance cycles       - (not selected)
        invalid end states      +

State-vector 72 byte, depth reached 124, errors: 0
     190 states, stored
      74 states, matched
     264 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573 memory usage (Mbyte)

unreached in proctype Agent
          (0 of 11 states)
unreached in proctype Server
        line 33, state 11, "client!deny,0"
        line 41, state 22, "-end-"
        (2 of 22 states)
unreached in proctype Client
        line 61, state 15, "-end-"
        (1 of 15 states)
```

Perhaps the one surprising detail in this result is that the statement on line 33, where the server summarily has to deny the request because its pool of available private channels is found to be empty, is not reachable. Given that the number of private channels in the server was defined to be equal to the number of clients, this result is easily understood. We can try to confirm our initial understanding of this phenomenon by increasing the number of client processes to three, without changing the number of channels declared in the server. Our expectation is now that the one unreachable statement in the server should disappear. This is the result:

```
$ spin -a client_server3.pml
```

```
$ cc -o pan pan.c
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
      never claim               - (none specified)
      assertion violations      +
      acceptance cycles         - (not selected)
      invalid end states        +

State-vector 84 byte, depth reached 331, errors: 0
     935 states, stored
     393 states, matched
    1328 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573 memory usage (Mbyte)

unreached in proctype Agent
        (0 of 11 states)
unreached in proctype Server
        line 33, state 11, "client!deny,0"
        line 41, state 22, "-end-"
        (2 of 22 states)
unreached in proctype Client
        line 62, state 15, "-end-"
        (1 of 15 states)
```

We can see that the number of reachable states increased, as expected given that we have more processes running in this system. But the statement on line 33 is still unreachable. What is going on?

Now we look more closely at the way in which we have defined the client processes. Note that a client process can only initiate a new request when `timeout` is true. This only happens if no other process in the entire system can make a step. This means that effectively only one of the three client processes will be executing in this system at a time. (Exercise: try to find a way to prove by model checking that this is true.) The three clients have different process identifiers, so each of the clients generates a different set of system states when it executes. The symmetry in this system is not automatically exploited by SPIN.

As an experiment, we can replace the `timeout` condition with `true`, and see if this helps to exercise the rogue statement. This is most easily done by adding a macro definition to the model, for instance, as follows:

```
#define timeout true
```

Repeating the verification run now produces the following surprising result:

```
$ ./pan
error: max search depth too small
pan: out of memory
```

```
        2.68428e+008 bytes used
        102400 bytes more needed
        2.68435e+008 bytes limit
hint: to reduce memory, recompile with
 -DCOLLAPSE # good, fast compression, or
 -DMA=652 # better/slower compression, or
 -DHC # hash-compaction, approximation
 -DBITSTATE # supertrace, approximation
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
...
```

The search ran out of memory. What happened?

By making it possible for the client processes to initiate requests at any time, we made it possible for a client to resubmit a request for service before the agent process that handled its last request has terminated. Consider, for instance, a request that was granted. After the client concludes the transaction by sending its return message, the agent process still has a number of steps to take before it can terminate. It must, for instance, first return the identifier of the now freed up private channel back to the server. If the client is fast enough, it can initiate a new transaction before the agent has completed the handling of its last transaction. This means that the process identificatio number of the last agent process cannot be recycled and reassigned for the new tranaction: the number of running agent processes can increase arbitrarily far. Most of these agent processes will eventually reach their termination state, where they could die. Because process death can only happen in stack order, the newly created agent processes now prevent the older processes from dying.

Though annoying, this potentially infinite increase in resource consumption does reflect a real hazard scenario that could also happen in a real system execution, so it is not without value. Our job as system designers is to find a way to make sure that this scenario cannot happen, by modifying the system design.

The best way to prevent the potential for runaway resource consumption is at the source: in the client processes. Sadly, there is no general rule for how this is best done: it will depend on the specifics of the model that one is using. In this case, we can easily make sure that no new client request can be submitted until the agent process for prior requests have terminated by replacing the `timeout` with a slightly more restrictive condition than `true`. The condition we will use in this model is as follows:

```
#define timeout (_nr_pr <= N+M)
```

The variable `_nr_pr` is a predefined system variable (see the manual pages) that gives the precise number of active processes in the model. How many processes should we maximally have in this model? There are `M` client processes, one server process, and maximally `N` agent processes. This gives an upper limit of `(N+M+1)` active processes. When a client is about to submit a new request, though, it should have no active agent process associated with itself anymore, so the maximum number of active processes in the system at the time that a new request is made should not be larger than `(N+M)`.

If we add this condition to the model and repeat the verification we see the following result:

```
$ ./pan -m30000
(Spin Version 4.0.7 -- 1 August 2003)
```

```
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance cycles       - (not selected)
        invalid end states      +

State-vector 108 byte, depth reached 26939, errors: 0
  133932 states, stored
  306997 states, matched
  440929 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 47515 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
15.536   equivalent memory usage for states
7.194    actual memory usage for states (compression: 46.30%)
         State-vector as stored = 46 byte + 8 byte overhead
1.049    memory used for hash table (-w18)
0.960    memory used for DFS stack (-m30000)
9.177    total actual memory usage

unreached in proctype Agent
        (0 of 11 states)
unreached in proctype Server
        line 41, state 22, "-end-"
        (1 of 22 states)
unreached in proctype Client
        line 63, state 15, "-end-"
         (1 of 15 states)
```

This is the result we were expecting when we tried to change `timeout` into `true`: all statements in the model, including the pesky statement on line 33, are now reachable, though at the expense of a considerable increase of the reachable state space.

As a general rule, when you see apparently infinite growth of the state space, signified by an apparently uncontrollable growth of either the state vector or the search depth, it is worth looking carefully at all the `run` statements in the model, to see if a scenario like the one we have discussed here is possible.

Square Roots?

We began our introduction to PROMELA in Chapter 2 almost inevitably with the PROMELA version of hello world. In retrospect, we can see that this example stretches the meaning of the term verification model. It defines only one single process, so clearly not much process interaction or synchronization could be happening. A model checker may be used to demonstrate that this little system cannot deadlock or get entangled into non−progress cycles, but the results that are obtained from such experiments will not be much of a revelation. Of course, PROMELA does not prevent us from writing such models, although it does try to deprive us from the tools we would need to put too much emphasis on non−concurrency aspects. This shows up specifically in the rudimentary support in the language for specifying pure computations. There are, for instance, no data types for `float`, `double`, or `real` in PROMELA. There is also no direct support for function calls or for recursion. These omissions are not accidental. It is deliberately hard to specify anything other than rudimentary computations in PROMELA, and deliberately easy to specify the infrastructure and the mutual dependency of concurrently executing processes.

This is not to say that no computation whatsoever can be done in PROMELA. It can. As a small example, consider the following PROMELA program that computes the integer square root of a given integer number. [1]

[1] The algorithm is due to Mark Borgerding. It appeared first at http://www.azillionmonkeys.com/qed/sqroot.htm.

```
proctype sqroot(int N)
{       int x, y;

        y = 1<<15;
        do
        :: y > 0 ->
                 x = x^y;              /* set bit   */
                if
                :: x*x > N ->    /* too large */
                          x = x^y      /* clear bit */
                :: else                /* leave set */
                fi;
                y = y>>1         /* next bit  */
        :: else ->
                break                  /* done      */
       od;
       printf("integer sqrt(%d) = %d\n", N, x)
}
```

A few constructs are used here that will look familiar to C programmers. The `proctype` named `sqroot` is declared non−active, which means that no instance of it is assumed to be started by default. An instance can be initiated by another process, and at the same time that process can then pass an integer parameter, `N`, to the newly instantiated process, specifying the number for which the integer square root is to be computed. That instantiation can look, for instance, as follows:

```
active proctype main()
{
        run sqroot(3601)
}
```

which uses the second mechanism in PROMELA to instantiate processes: through the use of the `run` operator.

Another, perhaps more convenient, way of defining the instantiation would be with a parameter, as in:

```
active proctype main()
{
        run sqroot(NR)
}
```

This allows us to experiment more easily with different inputs by invoking SPIN as follows:

```
$ spin -DNR=1024 sqroot.pml
integer sqrt(3601) = 55
2 processes created
```

SPIN reminds us that it created two process instances to execute this model: one called `main` and the other `sqroot`.

The body of `proctype sqroot` begins with the declaration of two integers. Since we do not have a floating point data type, using integers is the best we can do here. Unlike in C, the default initial value of all variables is zero in PROMELA, so both variables have a known initial value at this point. The variable `y` is assigned the value of an expression with one operator and two operands. The operator is the left−shift operator from C. This is a bit−operator that left−shifts the bit−pattern of the left operand the number of positions indicated by the right operand. If the left operand is one, as in this case, this amounts to computing the value of the expression 2^n, where n is the value of the right operand.

The next statement in the program is a loop. If the variable `y` is larger than zero, which upon first entry to the loop it will be, the first option sequence is executed. Once `y` reaches zero, the second option sequence will break from the loop, which causes the execution of the `printf` statement.

After the expression `y>0`, we see an assignment to variable `x`. The assignment computes the binary exclusive or of `x` and `y` and assigns it to `x`. The desired effect in this case is to set the bit position given by variable `y` (initially bit position 15, corresponding to a power of two).

The algorithm attempts to compute the value of the square root of `N` in variable `x`. It approaches the value from below, first looking for the highest power of two that can be included, without exceeding the target value. If the target value is exceeded, the bit position indicated by `y` is cleared again, by repeating the exclusive or operation; otherwise nothing needs to be done. The latter happens when the `else` in the selection statement becomes executable. Since nothing needs to be done, no statements need to be listed here. Some SPIN users feel uncomfortable about this, and would prefer to write the selection statements as follows,

inserting a dummy `skip` statement after the lonely `else`, but it is not required.

```
if
:: x*x > N ->           /* too large */
          x = x^y       /* clear bit */
:: else ->              /* leave set */
   skip
fi;
```

After the selection statement, the single bit in variable `y` is right−shifted by one position to derive the next lower power of two to be tested. When `y` reaches zero, the algorithm terminates and prints the computed integer value of the square root of `N`, as desired.

Because the entire computation is deterministic, there are many optimizations we could make in this little program to help SPIN execute it faster (e.g., by embedding the entire computation within an `atomic` or `d_step` statement), but we will leave well enough alone. The same observation that we made in the hello world example holds for this one, though: it would be a stretch to call this specification a verification model, since it defines no system design properties that could be verified with a model checker.

Adding Interaction

The main objection we can levy against the last example is that it really defines only sequential, not concurrent, behavior, with no synchronizations or interactions. With a few small changes we can turn the example into a slightly more interesting, though still rather minimal, distributed system model. We will set up the integer square root routine as a little square root server process that can perform its computations at the request of client processes. We will rearrange the code somewhat to accomplish this.

The first thing that we need to do is to declare a message channel for communication between clients and server, for instance, as follows:

```
#define NC 4

chan server = [NC] of { chan, int };
```

The first line defines a constant named `NC`. The constant is used in the declaration to set the capacity of the channel named `server`. The messages that can be passed through this channel are declared to have two fields: one of type `chan` that can hold a channel identifier, and one of type `int` that can hold an integer value.

Next, we rewrite the square root server process, so that it will read requests from this channel and respond to the client with the computed value, via a channel that the client provides in the request. The new version looks as follows:

```
active proctype sqroot()
{       chan who;
        int val, n;

        do
        :: server?who,n ->
             compute(n, val);
             who!val
        od
}
```

First, the server process declares a channel identifier named `who`, which this time is not initialized in the declaration. It also declares an integer variable `n`. These two variables are used to store the parameter values for the communication with a client, as provided by that client. A second integer variable `val` will be used to retrieve the result value that is to be communicated back to the client. The body of the square root server consists of a `do` loop with just one option, guarded by a message receive operation.

We have moved the actual computation into an `inline` definition, named `compute`. The variable name `n`, recording the value received from the client, is passed to the inline, as is the name of the variable `val` in which the result is to be computed.

After the call to `compute` completes, the value is returned to the client process in a send operation.

Before we fill in the details of the `inline` call, recall that an `inline` is merely a structured piece of macro text where the names of variables that are passed in as parameters textually substitute their placeholders inside the `inline` definition. Therefore, `inline`s are not the same as procedures: they cannot return results, and they do not have their own variable scope. All variables that are visible at the point of call of an `inline` are also visible inside the `inline` body, and, perhaps more noteworthy, all variables declared inside the `inline` are also visible outside it, after the point of call.

The inlined code for `compute` can now be written as follows:

```
inline compute(N, x)
{       int y;

        y = 1<<15;
        do
        :: y > 0 ->
                    x = x^y;             /* set bit */
                    if
                    :: x*x > N ->  /* too large */
                                x = x^y   /* clear bit */
                    :: else              /* leave set */
                    fi;
                    y = y>>1       /* next bit */
        :: else ->
                   break                 /* done */
        od;

}
```

All we need to complete the model is the code for the client process(es). The following declaration instantiates four distinct processes. Each has its own copy of a channel through which it can be reached by the server process.

```
active [NC] proctype client()
{       chan me = [0] of { int };
        int val;

        server!me,10*_pid ->
        me?val;
        printf("integer sqrt(%d) = %d\n", 10*_pid, val)
}
```

Each process multiplies its own process identifier, available in the predefined variable `_pid`, by ten and asks the square root server to compute the square root of the resulting number. It does so by sending this value, together with the value of the channel variable over which it can be reached through a send operation. Again, the types of the variables that make up the message sent must match the corresponding fields in the channel declaration for the channel addressed. In this case, it is a channel type, followed by an integer. The send operation is executable when the channel is non−full.

The private channel of the client process is declared in a slightly different way from the global channel named `server`. First, it has only one field, of type `int`. Next, the number of slots in this channel is declared to be zero to identify the channel as a rendezvous port that can pass messages in an atomic operation but cannot store or buffer messages. The receive operation on the client's rendezvous port is executable only if and when the server process reaches the send statement in its code. The client is blocked until this happens, but once it happens the client can print the value and terminate.

The complete model we have discussed is shown in Figure 15.7. Executing this model, using SPIN's default simulation mode, produces the following

Figure 15.7 Square Root Server Model

```
#define NC        4

chan server = [NC] of { chan, int };

inline compute(N, x)
{       int y;

        y = 1<<15;
        x = 0;                          /* reset x */
        do
        :: y > 0 ->
                    x = x^y;            /* set bit */
                    if
                    :: x*x > N ->   /* too large */
                                x = x^y     /* clear bit */
                    :: else             /* leave set */
                    fi;
                    y = y>>1        /* next bit */
        :: else ->
                    break               /* done */
        od;

}

active proctype sqroot()
{       chan who;
        int n, val;

        do
        :: server?who,n ->
                compute(n,val);
                who!val
        od
}

active [NC] proctype client()
{       chan me = [0] of { int };
        int val;

        server!me(10*_pid) ->
        me?val;
        printf("integer sqrt(%d) = %d\n", 10*_pid, val)
}
```

output:

```
$ spin sqroot_server
                    integer sqrt(30) = 5
      integer sqrt(10) = 3
                             integer sqrt(40) = 6
              integer sqrt(20) = 4
timeout
#processes: 1
              queue 5 (server):
481: proc 0 (sqroot) line 27 "srs" (state 19)
5 processes created
```

The output is somewhat more interesting this time. Each of the four client processes prints one line of output. To make it easier to keep track of consecutive output from individual processes, SPIN arranges for the output from each process to appear in a different column. Next, the system execution comes to a halt without all processes having been terminated. SPIN tries in this case to enable a timeout mechanism that can trigger further actions from the halted processes (we will see later how this is done). In this case, the `timeout` event triggers no further executions, and to wrap up the simulation SPIN reports for each non−terminated process at which line number and internal state is blocked. Only the square root server process is blocked, patiently waiting for further requests from clients, which in this model can no longer come.

If we wanted the server process to exit when the system comes to a halt, we could give it a timeout option as follows:

```
active proctype sqroot()
{  chan who;
  int n, val;

  do
  :: server?who,n ->
          compute(n,val);
          who!x
  :: timeout ->
          break
  od
}
```

With this extension, the final wrapup of the execution will reduce to just the note about the number of processes that was executed.

To see the communication inside this little process model, we can use some other SPIN options, for instance, as follows:

```
$ spin −b −c sqroot_server
proc 0 = sqroot
proc 1−4 = client

q\p   0   1   2   3   4
  5   .   .   .   server!3,30
```

```
  5   server?3,30
  5   .   .   server!2,20
  5   .   server!1,10
  5   .   .   .   .   server!4,40
  3   who!5
  3   .   .   . me?5
  5   server?2,20
  2   who!4
  2   .   .   me?4
  5   server?1,10
  1   who!3
  1   .   me?3
  5   server?4,40
  4   who!6
  4   .   .   .   .   me?6
5 processes created
```

The left hand column prints the channel number on which communication takes place, if any, and the top row gives the process pid numbers. Still more verbose output is also possible, for instance, by printing every single statement executed, or only specific types of statements. We will review the full range of simulation and verification options later in the book.

Adding Assertions

In the last version of the model we captured the behavior of a system of at least a few concurrent processes, and there was some interaction to boot. It is still not quite a verification model, though. The only thing we could prove about this system, for instance, is that it cannot deadlock and has no unreachable code segments. SPIN does not allow us to prove any mathematical properties of this (or any) square root algorithm. The reason is that SPIN was designed to prove properties of process interactions in a distributed system, not of process of computations.

To prove some minor additional facts about the process behaviors in this example, we can nonetheless consider adding some assertions. We may, for instance, want to show that on the specific execution of the square root computations that we execute, the result will be in the expected range. We could do so in the client processes, for instance, by modifying the code as follows:

```
active [NC] proctype client()
{       chan me = [0] of { int };
        int v;

        server!me(10*_pid) -> me?v;
        assert(v*v <= 10*_pid && (v+1)*(v+1) > 10*_pid)
}
```

Another thing we can do is to select a more interesting set of values on which to run the computation. A good choice would be to select a value in the middle of the range of integer values, and a few more that try to probe boundary cases. Since we cannot directly prove the mathematical properties of the code, the best we can do is to use one approach that resembles testing here. To illustrate this, we now change the client processes into a single tester process that is defined as follows:

```
active proctype tester()
{       chan me = [0] of { int };
        int n, v;

        if
        :: n = -1   /* fails */
        :: n = 0   /* ok */
        :: n = 1023   /* ok */
        :: n = 1<<29 /* ok */
        :: n = 1<<30 /* fails */
        fi;
        server!me(n) -> me?v;
        assert(v*v <= n && (v+1)*(v+1) > n)
}
```

Executing this model in SPIN's simulation mode as before may now succeed or fail, depending on the specific value for `n` that is chosen in the non−deterministic selection at the start of the tester. Along the way, it is worth observing that the five option sequences in this selection structure all consist of a single guard, and the

guards are all assignments, not conditional expressions. The PROMELA semantics state that assignments are always executable, independent of the value that is assigned. If we execute the little model often enough, for example, five or more times, we will likely see all possible behaviors.

Not surprisingly, the algorithm is not equipped to handle negative numbers as input; the choice of −1 leads to an assertion failure. All other values, except for the last, work fine. When the value `1<<30` is chosen, though, the result is:

```
$ spin -c sqroot_tester
proc 0 = sqroot
proc 1 = tester
q   0   1
  2   .   server!1,1073741824
  2   server?1,1073741824
  1   who!65535
  1   .   me?65535
spin: line 45 "srs2", Error: assertion violated
spin: text of failed assertion:
        assert((((v*v)<=n)&&(((v+1)*(v+1))>n)))
-------------
final state:
-------------
#processes: 2
                queue 2 (server):
108:    proc 1 (tester) line 45 "srs2" (state 9)
108:    proc 0 (sqroot) line 27 "srs2" (state 19)
2 processes created
```

It would be erroneous to conclude anything about the correctness of the algorithm other than for the specific values used here. For instance, it would not be safe to conclude that the algorithm would work correctly for all positive values up to `(1<<30)-1`.

Executing the simulation repeatedly by hand, until all cases in non−deterministic selection structures were hit, would of course be a tiresome procedure. The errors will come out more easily by simply running the verifier, for instance, as follows:

```
$ spin -a sqroot_tester
$ cl -DPC pan.c
$ pan
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance cycles       - (not selected)
        invalid end states      +

State-vector 92 byte, depth reached 39, errors: 1
      39 states, stored
       0 states, matched
```

```
      39 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573   memory usage (Mbyte)
```

The trail reveals the cause of the error. It is generated as follows:

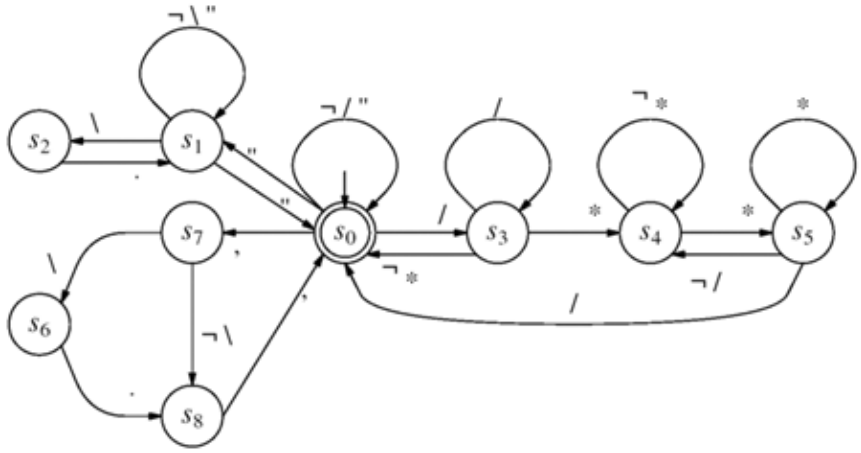
```
$ spin -t -c sqroot_tester
proc 0 = sqroot
proc 1 = tester
q   0   1
  2   .   server!1,-1
  2   server?1,-1
  1   who!0
  1   .   me?0
spin: line 72 "sqroot_tester", Error: assertion violated
spin: trail ends after 40 steps
-------------
final state:
-------------
#processes: 2
               queue 2 (server):
 40:    proc 1 (tester) line 73 "sqroot_tester" (state 11)
 40:    proc 0 (sqroot) line 28 "sqroot_tester" (state 21)
2 processes created
```

A Comment Filter

In Chapter 3 (p. 69) we briefly discussed a seldom used feature in PROMELA that allows us to read input from the user terminal in simulation experiments. The use of `STDIN` immediately implies that we are not dealing with a closed system, which makes verification impossible. Still, the feature makes for nice demos, and we will use it in this last example to illustrate the use of PROMELA `inlines`.

The problem we will use as an excuse to write this model is to strip C–style comments from text files. Figure 15.8 shows the general outline of a deterministic automaton for stripping comment strings from C programs. According to the rules of C, the character pair `/*` starts a comment and the first subsequent occurrence of the pair `*/` ends it. There are some exceptions to this rule though. If the combination `/*` appears inside a quoted string, for instance, it does not start a comment, so the automaton must be able to recognize not just comments but also quoted strings. To make things more interesting still, the quote character that starts or ends a string can itself be quoted (as in "'") or escaped with a backslash (as in "\""). In the automaton from Figure 15.8, states s_1 and s_2 deal with strings, and states s_6, s_7, and s_8 deal with quoted characters.

Figure 15.8. A Sample Automaton to Check C–Style Comment Conventions



The transition labels that we have used in Figure 15.8 represent classes of input characters that must be matched for the transition to be executable. The meaning is as follows. A transition label that consists of a single symbol, represents a match in the input stream of the corresponding ASCII character. A dot symbol, for example, on the transition from s_2 to s_1, represents a match of any single character in the input stream. The symbol ¬ (pronounced not) represents a match on any character other than all those that are listed behind the ¬ symbol in the transition label. This symbol is not itself an ASCII character, so there can be no confusion about its meaning. The label ¬ * on the transition from s_3 to s_4, for instance, represents the match of any character other than *, and the label ¬ /"on the self–loop at state s_0 means any character other than the forward slash character / or the quote character".

If the input sequence provided to this automaton conforms to the C comment conventions that are captured here, the automaton should terminate in its initial (and final) state s_0.

Part of the complexity of this automaton comes from the fact that characters can be escaped with backslashes, and can appear inside single or double quote marks. The comment delimiters could also appear inside text strings, making it hard to accurately recognize where a comment begins and ends in cases such as these:

```
/* the comment begins here
 * printf("but it doesn't end */ here yet\n");
 */
if (s == '"') /* not the start of a string */
        printf("/* not a comment */\n");
```

To simplify things a little, and to allow us to concentrate on the use of `inline`s in the example, we will cheat and not implement the full automaton but just a portion of it that implements the most basic functionality.

The basic model shown in Figure 15.9 can strip standard comments from text files, but does not make any exceptions for quoted or escaped characters, and it does not skip over text that is embedded in strings, as it formally should.

Figure 15.9 A Simple C–Style Comment Filter

```
1 /* strip C-style comments -- simple version */
2
3 chan STDIN;
4
5 #define FlipState in_comment = (in_comment -> false : true)
6
7 inline Print() {
8     if
9     :: !in_comment -> printf("%c", c)
10    :: else /* inside a comment string */
11    fi;
12 }
13 inline Getc(prev) {
14     do
15     :: STDIN?c ->
16             if
17             :: c == -1 -> goto done
18             :: c != prev -> break
19             :: c == prev -> Print()
20             fi
21     od
22 }
23 inline Handle(have, expect) {
24     oc = have;
25     Getc(have);
26     nc = c;
27     if
28     :: c == expect -> FlipState; goto again
29     :: else -> c = oc; Print(); c = nc
30     fi
31 }
32 init {
33     int c, oc, nc;
34     bool in_comment;
35
36 again: do
37     :: Getc(0) ->
38             if
39             :: !in_comment && c == '/' ->
40                     Handle('/', '*')
41             :: in_comment && c == '*' ->
```

```
42                      Handle('*', '/')
43              :: else
44              fi;
45              Print()
46      od;
47 done:         skip /* saw the -1 end-of-file marker */
48 }
```

As a quick test, if we run this program on itself, it nicely reproduces its own code, but without the comments. To avoid automatic indentation of the output, we can use SPIN's option `-T`. The command we execute is then:

```
$ spin -T strip < strip
```

which, in part, produces as output:

```
chan STDIN;

#define FlipState in_comment = (in_comment -> false : true)

inline Print() {
        if
        :: !in_comment -> printf("%c", c)
        :: else
        fi;
}

...

init {
        int c, oc, nc;
        bool in_comment;

again:  do
        :: Getc(0) ->
                if
                :: !in_comment && c == '/' ->
                        Handle('/', '*')
                :: in_comment && c == '*' ->
                        Handle('*', '/')
                :: else
                fi;
                Print()
        od;
done:   skip
}
1 process created
$
```

We use SPIN's `-T` option here to make sure that the printed characters are produced without indentation.

The main execution loop of the `strip` model appears in the `init` process. The boolean variable `in_comment` is initially false. It changes its value each time that the macro `FlipState` is called, which

uses PROMELA's syntax for a conditional expression to do so. An equally effective method would be to use the following definition:

```
#define FlipState in_comment = 1 - in_comment
```

but this relies perhaps a bit too much on the fact the SPIN normally does very little type checking.

The main execution loop in the initial process starts with an an `inline` call, using the parameter value zero. The inline reads a character from the magic `STDIN` channel, and checks whether it equals the predefined value for end−of−file: −1. Note that for this reason it is important to read the value into a signed integer here: using an unsigned `byte` variable would not work. If not the end−of−file marker, the value is compared with the parameter value that was passed in and if it does not match it, the read loop ends. In this case, the value read cannot be zero, so the loop will end on its first termination, leaving the value read in the integer variable `c`.

The next thing to do is to check for either the start or the end of a comment string, depending on the current state of variable `in_comment`. The one tricky part here is to correctly recognize the comment string in cases such as the following:

```
c//**** a strange comment ***/c
```

which, after stripping the embedded comment, should produce:

```
c/c
```

The `inline` function `Handle` checks for the expected character, but also passes the current character to `Getc` where it can now check for repetitions. Once the expected value is seen, the state of `in_comment` can change. If the character sequence turns out not to be a comment delimiter, though, we must be able to reproduce the actual text that was seen. For this reason, the `Handle` function uses the local variables `oc` and `nc`. Note that even though these two variables are only used inside the `Handle` function, the scope of all local variables is the same, no matter where in a process the declaration appears. Also, since the `Handle` function is called twice, we want to avoid inserting the declaration of these two variables twice into the body of the `proctype init`, which would happen if we moved the declaration into the `Handle` function itself.

The calls to `inline` function `Print` can appear up to three levels deep in an inlining sequence. For instance, `Handle` calls `Getc`, which can call `Print`. The calling sequence is valid, as long as it is not cyclic, since that would cause an infinite regression of inlining attempts. The SPIN parser can readily reject attempts to create a cyclic inlining sequences, as it requires that every `inline` function is declared before it is used.

We leave the extension of the model from Figure 15.9 to match the complete automaton structure from Figure 15.8 as an exercise.