Reliable Design of Concurrent Software

Gerard J. Holzmann
Computing Principles Research Department
Bell Laboratories 2C-521, Murray Hill, NJ 07974, USA
Email: gerard@research.bell-labs.com

ABSTRACT

Conventional static and dynamic testing techniques quickly break down if they are applied to distributed systems software. The bugs in such systems are usually triggered by irreproducible event sequences that can make debugging a nightmare. There are still tools that can help a programmer build a reliable system. One of the most popular such tools is the SPIN model checker. This article explains how this tool works, and what types of errors it can help you find.

1. Introduction

Whether we like it or not, we often design our code either by trial and error, or by duplicating and modifying a piece of code that does something similar to what we want. This, of course, works fine for small applications, but it fails miserably for large design projects or for critical code (e.g., code to control a nuclear power plant, or code to manage you bank account). Is there a better way? Before answering that question, consider how an architect goes about designing a structure. To remodel a shed or a small kitchen, no doubt he or she will improvise, but when a large structure has to be developed a quite different method is used. The methods that an engineer uses to design a bridge actually have a parallel in programming. The essential steps are to: (1) be a little more explicit about the requirements for a new design, and (2) to build design prototypes that can be checked for their compliance with these requirements. This may sound forbidding at first, but with the right tools all this can become quite straightforward, as we will try to show.

2. Model Checking

Just how can we build a prototype of a new program? Actually this is more common that you would imagine. It is precisely what authors do when they write about programming. To justify a particular programming procedure, the author gives a rather abstract *algorithmic* description of the program, often in a pseudo-language. All we have to do is to standardize this ‘‘pseudo-language,’’ develop a tool that can inspect the descriptions in this language, and compare them with a list of requirements. This procedure is called *model checking*. If applied well, the method can be amazingly effective in shaking out very subtle bugs in program designs.

The model checker SPIN comes with a specification language for quickly defining prototypes of distributed systems code, and a simple notation for defining the requirements on its correctness. There are two conditions that have to be fulfilled to make SPIN’s model checking procedure work: the model (i.e., the system prototype) is *finite state* and it is *closed*.

A system is *closed* if no part of the system behavior is left undefined. This is not true in general for sequential computer programs, since their specific behavior can depend on the inputs that it receives. To verify such a program, the input source has to be included in the model to close the system.

A system is *finite state* if it only has a finite number of potentially reachable states. The state of a system is simply the exhaustive set of all its data values and control points. If there are finitely many such state descriptions, a verifier could in principle perform any type of verification by enumerating the reachable states, and checking for the adherence to or violation of the design requirements.

Example New and old algorithms for achieving mutual exclusion in a distributed system, with a software procedure (i.e., without resort to special hardware instructions) provide a rich source of examples to demonstrate the power of model checking techniques. To get the flavor of this, let us look at one of the first algorithms that was suggested to solve this problem in 1966 [Hy66].

```
1  Boolean array b(0;1) integer k, i, j,
2  comment process i, with i either 0 or 1;
3  C0:   b(i) := false;
4  C1:   if k != i then begin
5  C2:   if not (b(j) then go to C2;
6        else k := i; go to C1 end;
7        else critical section;
8        b(i) := true;
9        remainder of program;
10       go to C0;
11       end
```

This algorithm was proposed as a simplification of a (correct) algorithm due to the Dutch mathematician Dekker, and published by Edsger Dijkstra in 1965 [Dij65]. In this description, the value of `j` is defined to be the opposite of `i`, i.e., equal to `1 - i`. The algorithm can be executed by two processes concurrently, sharing access to the global variables. The algorithm is meant to secure that the two processes cannot simultaneously execute the part labeled *critical section*.

It is a simple matter to turn this description into a model that has all the properties we desire, i.e., being finite state and closed. The model below is given in PROMELA, a Process Meta Language that is recognized by the SPIN tool. It looks as follows.

```
1  int cnt, k, b[2];   /* all variables are initially 0 */
2
3  active [2] proctype P()
4  {
5      assert(_pid == 0 || _pid == 1);
6
7  C0: b[_pid] = 0;
8  C1: if
9      :: (k != _pid) ->
10 C2:          (b[1 - _pid]);  /* wait for this to be nonzero */
11             k = _pid;
12             goto C1
13
14     :: else ->
15             cnt = cnt+1;
16             assert(cnt == 1); /* the critical section */
17             cnt = cnt-1;
18
19             b[_pid] = 1;
20     fi;
21     goto C0
22 }
```

The translation is pretty close to the original, with some minor syntactical differences for specifying control-flow structures. The `if .. fi` construct from PROMELA is based on an older notation for specifying non-deterministic *guarded commands*, also due to Dijkstra [Dij75], that turns out to be eminently suitable for the description of high level models of distributed software. Unlike a normal programming language, the options in the `if .. fi` selection construct need not be mutually exclusive. If more than one option is selectable, the choice between them is non-deterministic.

To check the properties of this algorithm, we have to be a little more explicit about the requirements for its correctness. We have done so in this model by adding two *assertions* to the model. The first one, on line 5, merely checks that the processes have identifiers (predefined in PROMELA as `_pid`) that match our assumptions. If the `_pid` is either 0 or 1, we can obtain the `_pid` of the competing process with `1 - _pid`. The

second assertion, on line 16, states the actual mutual exclusion requirement. If mutual exclusion is correctly maintained, the value of the variable `cnt` can never exceed one.

It takes SPIN a fraction of a second to establish that the first assertion is valid, but the second is not: the mutual exclusion property can be violated. Now all this would still be of fairly little use unless the tool is able to tell us just exactly *how* the violation can occur. SPIN does so by generating an execution sequence for the model that leads from the initial program state to the violation of the assertion on line 16. There are a number of different ways to explore the sequence. The default is to have it printed as a simple interleaving sequence, displayed on an ASCII terminal as follows:

```
  1:    proc  1 (P) line   5 "hy66" (state 1)  [assert(((_pid==0)||(_pid==1)))]
  2:    proc  1 (P) line   7 "hy66" (state 2)  [b[_pid] = 0]
  3:    proc  1 (P) line   9 "hy66" (state 3)  [((k!=_pid))]
  4:    proc  0 (P) line   5 "hy66" (state 1)  [assert(((_pid==0)||(_pid==1)))]
  5:    proc  0 (P) line   7 "hy66" (state 2)  [b[_pid] = 0]
  6:    proc  0 (P) line  14 "hy66" (state 7)  [else]
  7:    proc  0 (P) line  15 "hy66" (state 8)  [cnt = (cnt+1)]
  8:    proc  0 (P) line  16 "hy66" (state 9)  [assert((cnt==1))]
  9:    proc  0 (P) line  17 "hy66" (state 10) [cnt = (cnt-1)]
 10:    proc  0 (P) line  19 "hy66" (state 11) [b[_pid] = 1]
 11:    proc  1 (P) line  10 "hy66" (state 4)  [(b[(1-_pid)])]
 12:    proc  0 (P) line   7 "hy66" (state 2)  [b[_pid] = 0]
 13:    proc  0 (P) line  14 "hy66" (state 7)  [else]
 14:    proc  1 (P) line  11 "hy66" (state 5)  [k = _pid]
 15:    proc  1 (P) line  14 "hy66" (state 7)  [else]
 16:    proc  1 (P) line  15 "hy66" (state 8)  [cnt = (cnt+1)]
 17:    proc  1 (P) line  16 "hy66" (state 9)  [assert((cnt==1))]
 18:    proc  0 (P) line  15 "hy66" (state 8)  [cnt = (cnt+1)]
spin: line  16 "hy66", Error: assertion violated
 19:    proc  0 (P) line  16 "hy66" (state 9)  [assert((cnt==1))]
spin: trail ends after 19 steps
#processes: 2
                cnt = 2
                k = 1
                b[0] = 0
                b[1] = 0
 19:    proc  1 (P) line  17 "hy66" (state 10)
 19:    proc  0 (P) line  17 "hy66" (state 10)
2 processes created
```

The trace can also be made more informative by adding the printouts of variable values and message channel contents (not used in this example) at every step. A front-end tool XSPIN to SPIN can also display the sequence graphically, or it can separate the actions of the two processes, and print the sequence in two columns as follows:

```
_pid:    0        1
         |        |>assert(((_pid==0)||(_pid==1)))
         |        |>b[_pid] = 0
         |        |>((k!=_pid))
         |>assert(((_pid==0)||(_pid==1)))
         |>b[_pid] = 0
         |>else
         |>cnt = (cnt+1)
         |>assert((cnt==1))
         |>cnt = (cnt-1)
         |>b[_pid] = 1
         |        |>(b[(1-_pid)])
         |>b[_pid] = 0
         |>else
         |        |>k = _pid
         |        |>else
         |        |>cnt = (cnt+1)
         |        |>assert((cnt==1))
         |>cnt = (cnt+1)
         |>assert((cnt==1))        # assertion violated
```

In any case, it takes considerably less effort to show that the design is flawed than it took the creator of the algorithm to argue the opposite.

The model we built for the mutual exclusion algorithm trivially has the two required properties for model checking techniques to apply: it defines a closed system, with no undefined external stimuli, and it defines a finite state system, with only finitely many control states per process and finitely many possible values per control variable used.

3. Capabilities

The advantage of the model checking technique is that if the tool completes its search for bugs and comes back empty-handed this will actually guarantee that all explicitly stated requirements that are satisfied for the model: a much stronger result than can be achieved with any other debugging technique. Of course, there are also disadvantages. One is that it is not easy to extend the model checking technique to debug models for arbitrary (and possibly infinitely) many system components. Another is that showing the correctness of the major design decisions captured in a model does not necessarily guarantee that no new bugs are introduced when the model is implemented. As always, there's no free lunch. The model checking tools can only help the programmer to rule out one class of errors that can be hard, if not impossible, to find with standard static or dynamic testing techniques.

Model checkers such as SPIN are not restricted to checking simple state assertions such as the one used in the example. Basically, any type of requirement on valid or invalid executions of the system can be formalized in the underlying logic of the model checker. The underlying logic is linear temporal logic, LTL for short. The logic contains simple operators to state requirements of sequences of conditions that ought to be satisfied, e.g., that once a process tries to enter its critical section 'eventually' it cannot fail to gain access. (Not true for the example algorithm, as SPIN can show with an example in a fraction of a second. To appreciate this capability, try to find an execution sequence that denies one of the processes entry to the critical section forever...)

There have been many applications of the SPIN model checker to problems of considerable size. SPIN has found bugs in distributed algorithms, protocol designs, railway signaling methods, process scheduling algorithms, telephone switching code, etc. SPIN is not just for big design problems, though, it's a lot of fun to play with on little problems too.

4. Getting the Code.

SPIN and XSPIN are available freely via URL `http://netlib.bell-labs.com/netlib/spin`, documentation and sample applications included. For further reading on SPIN, see e.g., [Ho91], [Hol97].

5. References

[Dij65] Dijkstra, E.W., Solution of a problem in concurrent programming control, *Comm. of the ACM*, 1965, Vol. 8, No. 9, p. 569.

[Dij75] Dijkstra, E.W., Guarded commands, nondeterminacy and formal derivation of programs, *Comm. of the ACM*, 1975, Vol. 18, No. 8, p. 453-457.

[Ho91] Holzmann, G.J., *Design and Validation of Computer Protocols*, Prentice Hall, Software Series, 1991.

[Ho97] Holzmann, G.J., The model checker SPIN, *IEEE Trans. on SE*, 1997, Vol. 23, No. 5. (Special issue on formal methods in software practice, to appear.)

[Hy66] Hyman, H. Comments on a problem in concurrent programming control, *Comm. of the ACM*, 1966, Vol 9, No. 1, p. 45.

Figure Captions (Screen Dumps, as attached)

1. SPIN's main control window, showing an annotated version of the mutual exclusion example discussed in the paper. SPIN is written in ANSI standard C, and runs on any Unix system and Windows95 PC's. XSPIN, SPIN's graphical interface, is a simple Tcl/Tk application, independent of SPIN itself, that gives easy access to all of SPIN's functionality, and that can automate routine chores.
2. SPIN works internally by computing a partial product of automata that it compares against a temporal formula that contains the requirements. The automata that SPIN generates behind the scenes can be made visible with XSPIN. This figure shows the automaton for one of the processes in the mutual exclusion example.
3. SPIN reports bugs as execution sequences. By default, these sequences are printed in ASCII. This figure shows an alternative provided by XSPIN, to show sequences graphically (in this case for a leader election protocol in a distributed ring). By hovering the cursor over the chart the corresponding source line in the main SPIN control window is highlighted.