SPIN as a Hardware Design Tool

Budi Rahardjo
Electrical and Computer Engineering
University of Manitoba
15 Gillson Street
Winnipeg, MB, Canada - R3T 5V6
E-mail: `rahard@ee.umanitoba.ca`

Abstract

This paper presents the application of SPIN to verify a hardware design. A hazardoues circuit is modelled in PROMELA and verified with SPIN. SPIN shows the presense of hazard. The circuit was corrected and verified.

Keywords: *hardware design and verification, protocol engineering, CASE tools, PROMELA, SPIN.*

1 Introduction

One engineering approach to design and analyze a large and complex hardware design is to decompose the design into modules or sub-designs, each of which can be developed and verified individually. In the final design, these modules interact or communicate with each other with a set of predefined rules: a protocol. While individual modules may have been proven to be correct, the composition of them does not necessarily exhibit the expected behavior. The design must still be validated or verified.

In this paper, we argue that tools to validate computer protocols can be used to verify hardware designs. We illustrate this by modelling a simple circuit in PROMELA, a language to model computer protocols developed by Holzmann [3], and verified with an automated tools called SPIN [3].

The use of protocol tools to aid hardware designers has been proposed in [1]. Another relation between protocol tools (SDL CASE tool in this case) and hardware design is reported in [2].

2 A hazardous circuit

In this paper we show the use of SPIN to find hazards in a *fundamental-mode* circuit, i.e., the circuit must be stabilized completely before another input change can be applied.

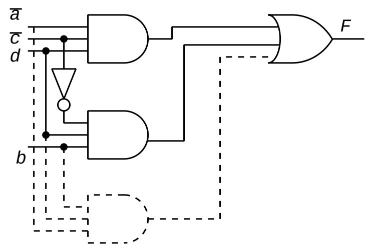

Figure 1: A hazardous fundamental-mode circuit

cd	ab 00	ab 01	ab 11	ab 10
00				
01	1	1		
11		1	1	
10				

Figure 2: K-map

Figure 1 shows a circuit that contains a hazard (ignore the dashed lines and gate.) Its K-map is shown in figure 2 (ignore the dashed ellipse). The circuit has the following Boolean function:

$$F = \bar{a}\bar{c}d + bcd \tag{1}$$

The following explanation will show that the implementation is not a hazard-free one. Depending on the delays of the inverter and wires, during a transition on signal or wire c, there is a possibility that a spike is generated. A possible scenario is shown in Table 1. In this situation, $\bar{c}$ (indicated as `cbar` in the table) is changing from “1” to “0” while the other input signals are still constant. As shown in row 5 and 6 of Table 1, there is a momentary dip, or hazard, at the output (F).

Table 1: Hazardous transitions

row#	abar	cbar	b	d	n1	n2	c	F
1	1	1*	1	1	1	0	0	1
2	1	0	1	1	1*	0	0	1
3	1	0	1	1	0	0	0*	1
4	1	0	1	1	0	0	1	1*
5	1	0	1	1	0	0*	1	0
6	1	0	1	1	0	1	1	0*
7	1	0	1	1	0	1	1	1

* = about to change

3 PROMELA model

In this section we describe a possible PROMELA model or description of our circuit. (For a complete description of the syntax and semantics of PROMELA, we refer the reader to [3].)

```
/* declaration */
bit a, abar, b, c, cbar, d, f;
bit oldf;
bit n1, n2;
bit newp;

#define AND3(x,y,z,out)  (out != (x&&y&&z)) -> out = x&&y&&z
#define OR2(x,y,out)     (out != (x||y)) -> out = x||y
#define OR3(x,y,z,out)   (out != (x||y||z)) -> out = x||y||z
#define INV(in,out)      (out != (1-in)) -> out = (1-in)


proctype netlist()
{
   do
   ::
      if
      :: AND3(abar, cbar, d, n1);
      :: INV(cbar, c);
      :: AND3(b, c, d, n2);
      :: OR2(n1,n2,f);
      fi;
      newp=0
   od
}

proctype stimulus()
{
   do
   :: timeout ->
      atomic {
      newp=1;
      oldf=f;
      if
      :: abar = 1-abar
      :: b = 1-b
      :: cbar = 1-cbar
      :: d = 1-d
      fi;
      }
   od
}
init
{
   atomic{abar=0; cbar=0; b=0; d=0; newp=1 };
   atomic {
      run stimulus();
      run netlist()
      }
}

never {
   do
```

```
    :: skip
    :: (newp==0 && oldf != f) -> break /* transition */
    od;
    do
    :: ((newp==0) && (oldf != f))
    :: ((newp==0) && (oldf == f)) -> break /* spike/dip */
    od
}
```

The description is similar to a C program. It is started with a declaration and macro definitions. It is then followed by the circuit implementation, i.e., connections of the gates. This is done by using a process (indicated by the `proctype` keyword) called `netlist()` (name unimportant). Double colons (`::`) indicate selection and the `do-od` construct indicates repetition. The `if-fi` construct non-deterministically selects one of the selections. The `netlist` description indicates that on each cycle one of the gates is selected and evaluated until all gates are stable and the process is blocked.

The next description is a process called `stimulus()`. (Again, name is not important.) This process is started with the keyword `timeout`, which is set to FALSE until there is no more action on the model, at which time it will be set to TRUE. When it is TRUE, a new pattern is applied by toggling one of the inputs (under `if-fi` construct.) The `newp` flag is set to indicate that a new input pattern has been applied at this cycle. This technique implements the *fundamental mode* operation of the circuit, i.e., it will wait until the circuit stabilized and then change the input(s). All processes are started in parallel in the main block indicated by the `init` keyword.

The last part needed is the claim or specification that we want to verify. In our particular case, we want to express the non existence of hazards by using a *claim FSM*, indicated by the `never()` keyword. It is used to expressed behavior that should not happen. The explanation of this claim FSM is as follows. The “`skip`” is important. If it is omitted, then the proposition that we are trying to match must match the first reachable state. This is not what we want. We want to check the correctness regardless what the initial reachable state is. At some non-deterministic time, the proposition $((newp == 0)$ && $(oldf! = f))$ can be matched, i.e., no new input but there is a transition at the output. We then move (`break`) to the second `do-od` loop. One of two conditions may occur: a new pattern is applied (which will set $newp = 1$), or the output changes back ($oldf == f$) while input stays the same. In the former, a new input pattern is applied, and then search through this path is truncated, i.e., the circuit does not produce a hazard. In the latter, a hazard is detected and the temporal claim is completed, which means that the never claim is matched and an incorrect behavior is detected. The above claim corresponds to the existence ofa *static hazard* [4]. However, since a *dynamic hazard* must “pass” through a static hazard, it will also be detected.

4 Verification Results

The complete model was fed to SPIN. A C file was generated, compiled, and executed on a UNIX workstation.

```
unix% spin -a hazard.spin
unix% gcc -o pan pan.c
unix% ./pan
```

The execution was terminated and a hazard was found. The final state shows the following values. (Compare these values to the last row of Table 1.)

```
  n1 = 0
  n2 = 1
  _p = 0
  abar = 1
  cbar = 0
  oldf = 1 [previous value of F]
  a = 0
  b = 1
  newp = 0
  c = 1
  d = 1
  f = 1    [value of F]
```

Readers can use SPIN to trace the execution and monitor the variables.

5 Hazard Removal

A common method to remove hazards is by adding redundant terms. In this particular circuit, we can add another term indicated by the dashed lines and gate in Figure 1. With the addition of this term, the Boolean function becomes

$$F = \bar{a}\bar{c}d + bcd + \bar{a}bd \tag{2}$$

This additional term corresponds to the dashed line in the K-map shown in Figure 2. It holds the output during a transition $0 \rightarrow 1$ on wire c.

The additional term was added to the original PROMELA model and was verified with SPIN through full state space reachability analysis. No hazard was found.

6 Concluding Remarks

We have illustrated the use of PROMELA to model a hardware design and SPIN to verify it. We showed a technique to implement fundamental mode operation. This approach is not limitted to PROMELA. It applies to other protocol languages (such as SDL or Lotos) and tools. This approach has been tested in the verification of asynchronous circuits [6, 7, 5].

By performing protocol verification on a hardware design, we view the design at a higher level of abstraction. This is useful for complex designs, such as in designs that use complex data structures to communicate among (sub)modules. Instead of dealing with bits-and-bytes, one can deal with abstract messages. In our opinion, this is where high-level CASE tools, such as those commonly used in the validation of computer protocols, play important roles (e.g. in visual animation or simulation and verification.) We hope to see more integration between these tools and commercial VLSI CAD tools.

References

[1] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design*, pages 522–525, 1992.

[2] W. Glunz and G. Venzl. Hardware design using CASE tools. In A. Halaas and P. B. Denyer, editors, *VLSI 91*, 1992.

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

[4] G. M. Jacobs. *Self-Timed Integrated Circuits for Digital Signal Processing*. PhD thesis, Electrical and Computer Science, University of California at Berkeley, Dec. 1989.

[5] B. Rahardjo. Design and analysis of a counterflow pipeline processor in SDL. Technical report, Telecommunication and Research Laboratories, Winnipeg, Canada, 1995. (in preparation).

[6] B. Rahardjo and R. D. McLeod. Verification fo speed-independent asynchronous circuits with protocol validation tools. In *Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signa Processing (to appear)*, 1995.

[7] B. Rahardjo, J. F. Peters, and R. D. McLeod. Communicating processes in designing asynchronous circuits. *IEEE Aerospace and Electronic*, 10(7):8–11, July 1995.