Modeling the *Observer-Reporter* Problem in SMV

This problem is often used in operating systems textbooks to introduce concepts of *atomicity.* An *Observer* process is monitoring asynchronous events. When it senses that an event has occurred it increments a counter variable. In pseudocode:

```
Observer: while true do
            begin
            await EVENT;
            COUNT := COUNT + 1;
            end;
```

A *Reporter* process periodically prints the event count and clears the counter.

```
Reporter: while true do
            begin
            wait INTERVAL seconds;
            print(TIME, COUNT);
            COUNT := 0;
            end;
```

These higher level instructions must be compiled into a ”machine-level” model whose atomic instructions can only:

- MOV *source, destination* copies a unit (word or byte) of data to/from memory from/to a processor register. The MOV instruction does not alter the data.
- OP *opnd1, opnd2, result* performs an arithmetic operation on processor registers, leaving the result in a register.
- JMP *program-label* transfers control to another point in the program.

Using SMV models, let us explore whether the more finely grained machine-level implementation might miss events or count them more than once. Possible partial implementations of *Observer* and *Reporter* are shown below, in pseudo-assembly code:

```
OBSERVER: {wait for EVENT to occur, then}
          MOV count, r1   -- retrieve the value of COUNT
          ADD 1, r1, r1   -- increment it
          MOV r1, count   -- store the result
          JMP OBSERVER

REPORTER: {wait for an INTERVAL of time}
          MOV count, r1   -- retrieve the value of COUNT
          MOV 0, count    -- clear the global value
          Print r1      -- log the value
          JMP REPORTER
```

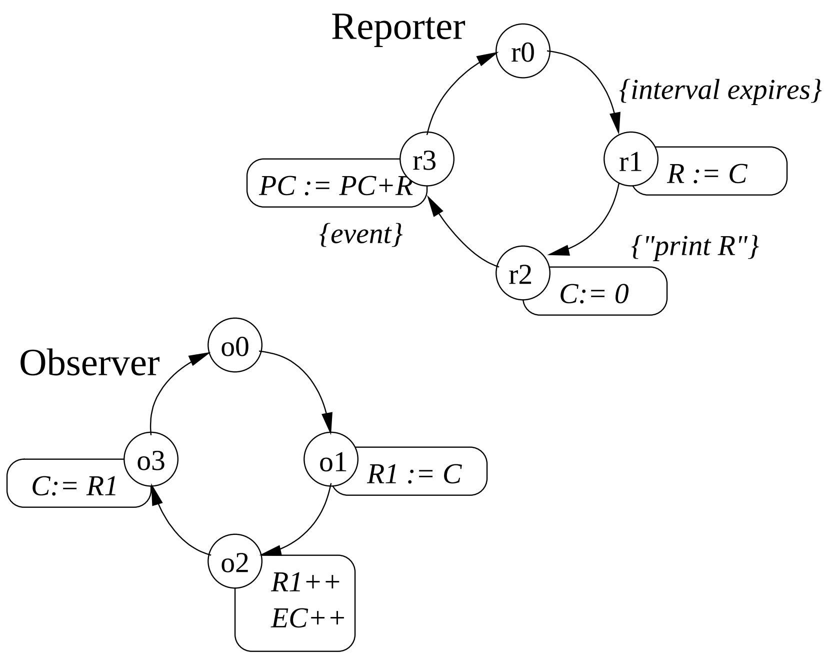


Figure 1: A model for the *Observer-Reporter* processes

This is what we want to model; now we need to express that model in SMV. The FSAs in Figure 1 suggest one way to do it.

The purpose of the model is to determine whether each of the events is reported exactly once. With that in mind, introduce and initialize three variables to represent the shared `count` variable, to tally the number of events seen by the *Observer*, and to record the number of events printed by the *Reporter*:

```
VAR
count   : 0..7; -- Count shared between Rep and Obs
print   : 0..7; -- Number of events reported
event   : 0..7; -- Number of events that actually occur

ASSIGN
init(event) := 0;
init(print) := 0;
init(count) := 0;
```

We have restricted the model substantially by limiting the range of the counters to 7. Since SMV models are finite, we must impose some limit, but we are guessing that whether 7 is large enough to expose the properties of interest, namely the relationship between `print`, the total number of events reported, and `event`, the total number of events that have occurred. The system composes two processes, `Obs` and `Rep`:

```
VAR
Obs     : process observer(count, event);
Rep     : process reporter(count, print);
```

If the system is correct, these totals should agree. But the specification

```
SPEC AG( print = event )
```

that says, "`print` *always* equals `event`," is false because it is too strong: these values change during the execution of the two loops. We want to compare them only when the processes are at the "top" of their loops. Furthermore, those events tallied in the `count` variable haven't been reported yet. Define `ready` to express this condition.

```
DEFINE
ready := Obs.ready & Rep.ready & (count = 0);
```

Conditions `Obs.ready` and `Rep.ready` are `DEFINE`d later in the respective `MODULES` instatantiated by `Obs` and `Rep`. The correctness property, then, is

```
SPEC -- 1. Whenever the system is ready, the event count is correct.
  AG(ready -> (print = event))
```

The model checker (GUI) will show this specification to be false.

Modules

NOTE: *The modules are discussed in greater detail in lecture*
The *Observer-Reporter* system consists of three modules. The `main` module composes one instance each of the `observer` (`Obs`) and `reporter` (`Rep`) modules defined later in the file. Modules may have parameters and local variables.

The `observer` module is Figure 4. has two local variables: Variable `pc` represents the four states of the *observer* FSA in Figure 1: `O0`, `O1`, `O2`, and `O3`. Variable `r` represents the single register used in its "assembly" program.

These values are globally accessable as `Obs.pc` and `Obs.r`, respectively, as is the local `ready` condition `Obs.ready`, used earlier, and defined as

```
DEFINE ready := pc = O0;
```

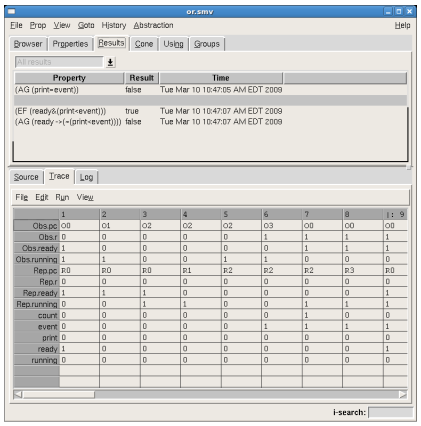
As the diagrams depict, `r` is "loaded" with the value `c` (bound to the global `count` in instance `Obs`) in state `O1`, incremented in state `O2`.

Similarly `r`'s value is moved to `c` state `O3`.

In state `O2` the `event` tally is incremememented.

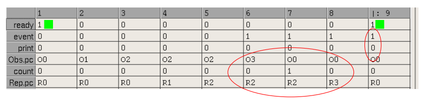
In `Obs`, taking a step means that *all* of the `next(`x`) :=` y assignments are performed simultaneously.

The `reporter` module in Figure 5 is similar.


or.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
All results
Source Trace Log
File Edit Run View
i-search:

Property	Result	Time
(AG (print=event))	false	Tue Mar 10 10:47:05 AM EDT 2009
(EF (ready&(print<event)))	true	Tue Mar 10 10:47:07 AM EDT 2009
(AG (ready ->(~(print<event))))	false	Tue Mar 10 10:47:07 AM EDT 2009

	1	2	3	4	5	6	7	8	|: 9
Obs.pc	O0	O1	O2	O2	O2	O3	O0	O0	O0
Obs.r	0	0	0	0	0	1	1	1	1
Obs.ready	1	0	0	0	0	0	1	1	1
Obs.running	1	1	0	0	1	1	0	0	0
Rep.pc	R0	R0	R0	R1	R2	R2	R2	R3	R0
Rep.r	0	0	0	0	0	0	0	0	0
Rep.ready	1	1	1	0	0	0	0	0	1
Rep.running	0	0	1	1	0	0	1	1	1
count	0	0	0	0	0	0	1	0	0
event	0	0	0	0	0	1	1	1	1
print	0	0	0	0	0	0	0	0	0
ready	1	0	0	0	0	0	0	0	1
running	0	0	0	0	0	0	0	0	0



	1	2	3	4	5	6	7	8	|: 9
ready	1 🟩	0	0	0	0	0	0	0	1 🟩
event	0	0	0	0	0	1	1	1	1
print	0	0	0	0	0	0	0	0	0
Obs.pc	O0	O1	O2	O2	O2	O3	O0	O0	O0
count	0	0	0	0	0	0	1	0	0
Rep.pc	R0	R0	R0	R1	R2	R2	R2	R3	R0

Figure 2: Default (left) and restricted-view VW screenshots

Interpreting Counter-Scenarios

When a `SPEC` statement is false, SMV gives counterexample, demonstrating a *scenario* in which the specification property fails. These scenarios give a step-by-step execution trace, often containing a great deal of irrelevant information.

To explain the results, one should reduce the verbose counter-scenario to show as clearly as possible those steps and relevant variables that lead to the "failure." For example, the sequence diagram in Figure 3[1] shows just those steps and actions that cause `count` to differ from `print` when the `ready` condition holds.

The `man` page for `vw` describes commands for organizing and filtering traces in the GUI.

[1]The scenario in Figure 3 may not be the same counterexample generated by `smv` when you run it.

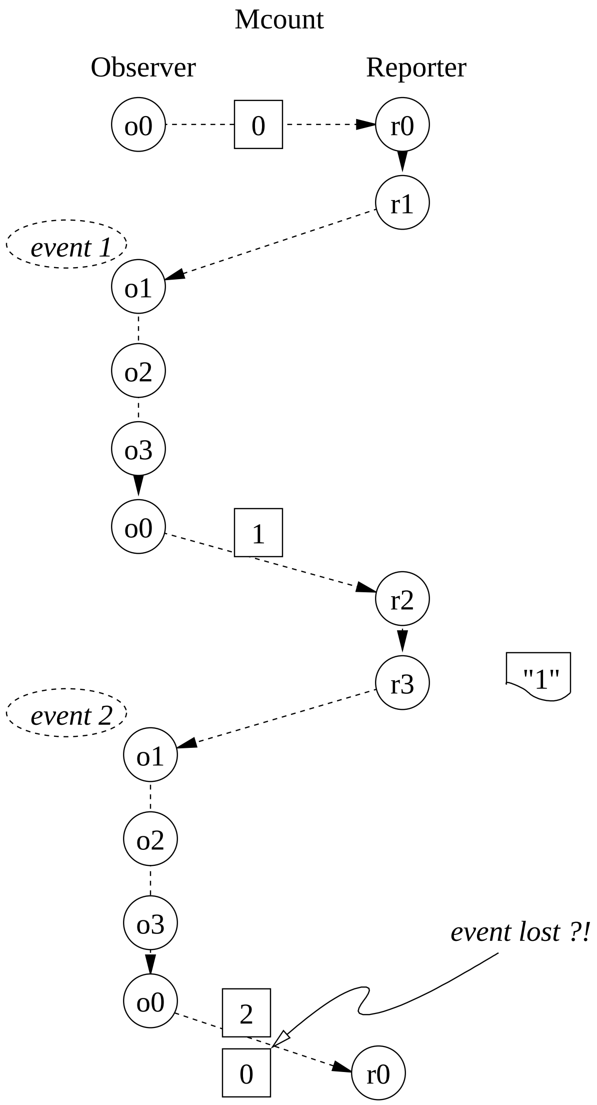


Figure 3: A counter-scenario refuting `AG( ready -> (count = print))`.

```
MODULE observer(c, e)

VAR
pc : {00,01,02,03};
r  : 0..7;            -- local register

DEFINE
ready := pc = 00;

ASSIGN
init(pc) := 00;
next(pc) :=
 case
  pc = 00  : 01;
  pc = 01  : 02;
  pc = 02  : 03;
  pc = 03  : 00;
  1        : pc;
 esac;

init(r) := 0;
next(r) :=
 case
  pc = 01  : c;
  pc = 02  : r + 1;
  1        : r;
  esac;

next(c) :=
  case
   pc = 03 : r;
   1       : c;
  esac;

next(e) :=
  case
   pc = 02 : e+1;
   1       : e;
  esac;

FAIRNESS running
```

Figure 4: SMV description of the *Observer* FSM in Figure 1

```
MODULE reporter(c, p)

VAR
pc : {R0, R1, R2, R3};
r: 0..7;

DEFINE
ready := pc = R0;


ASSIGN
init(pc) := R0;
next(pc) :=
 case
  pc = R0: R1;
  pc = R1: R2;
  pc = R2: R3;
  pc = R3: R0;
  1       : pc;
 esac;

next(r) :=
 case
  pc = R1: c;
  1      : r;
 esac;

next(c):=
 case
  pc = R2: 0;
  1      : c;
 esac;

next(p):=
 case
  pc = R3 : p + r;
  1       : p;
 esac;

FAIRNESS running
```

Figure 5: SMV description of the *Reporter* FSM in Figure 1