
CIS 721 - Real-Time Systems

Lecture 25: Hardware Model Checking

Mitch Neilsen
neilsen@ksu.edu

Outline

- Real-Time Verification and Validation Tools
 - UPPAAL – Toolbox for validation and verification of real-time systems
 - **Promela and SPIN**
 - Simulation
 - Verification
 - **Real-Time Extensions:**
 - **RT-SPIN – Real-Time extensions to SPIN**
-

Properties to Check using SPIN

- Deadlock
 - Livelock, starvation
 - Underspecification – Unexpected reception of messages
 - Overspecification – Dead code
 - Violations of constraints
 - Buffer overruns
 - Array bounds violations
 - No assumptions are made about speed; e.g., testing **logical correctness** versus **real-time behavior**
-

Promela

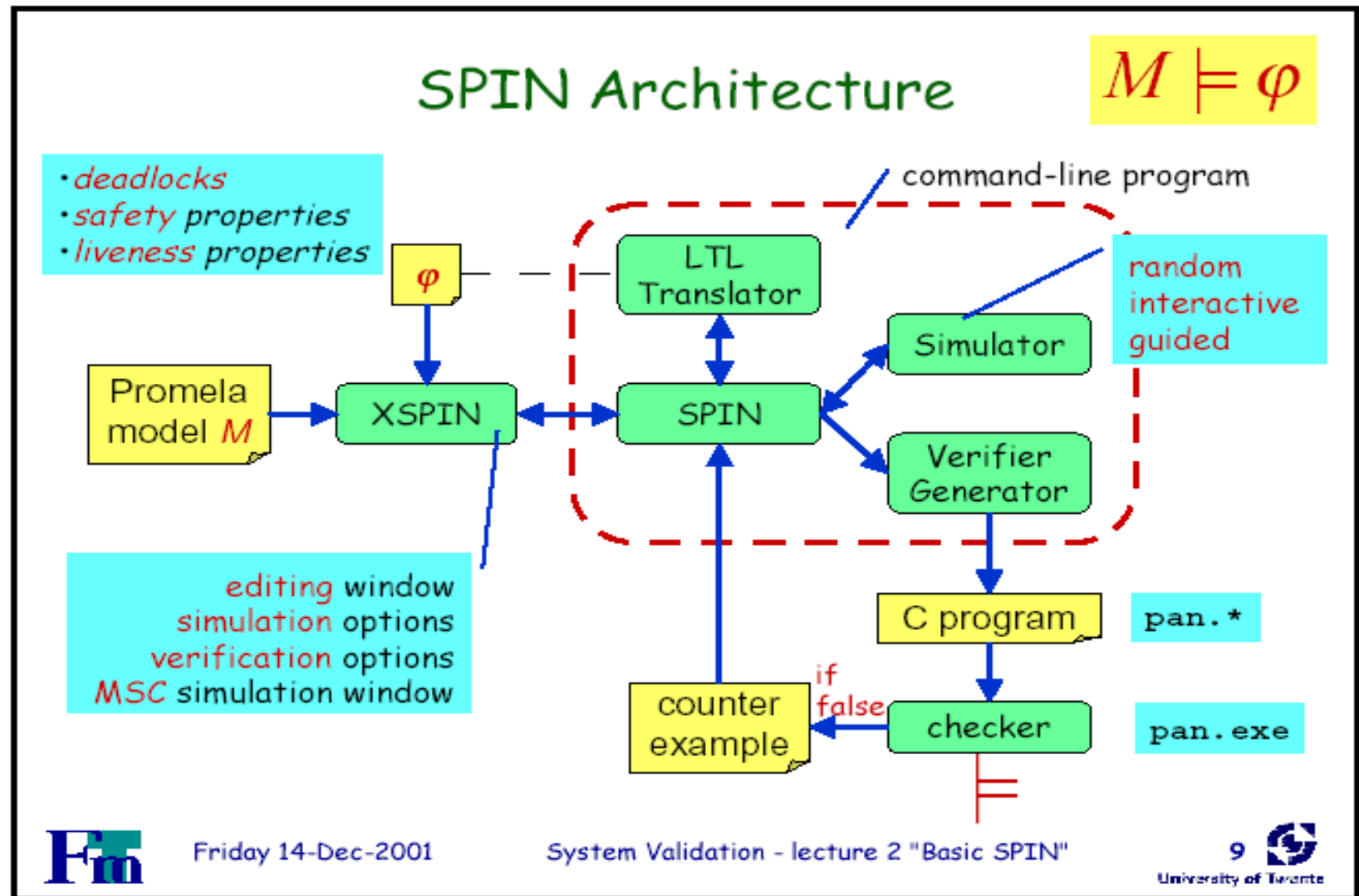
■ Promela – Process/Protocol Meta Language

- ❑ Provides a language similar to the C programming language
- ❑ Provides a guarded command language to model finite-state systems
- ❑ Supports dynamic creation of concurrent processes
- ❑ Supports messages channels between processes

SPIN

- **SPIN** – Simple Promela Interpreter
 - A state-of-the-art model checking tool used to check the logical consistency of concurrent systems described using the modelling language Promela.
 - Designed specifically for checking data communication protocols.
-

SPIN Architecture



State Vector

- A state vector is the information to uniquely identify a system state; it contains:
 - global variables
 - contents of the channels
 - for each process in the system:
 - local variables
 - process counter of the process
 - For efficient modelling, it is important to minimize the size of the state vector.
-

SPIN Algorithm

- SPIN uses a depth first search algorithm (DFS) to generate the complete state space (Statespace).

```
procedure dfs(s: state) {  
    add s to Statespace;  
    if error(s) reportError();  
    foreach (successor t of s) {  
        if (t not in Statespace)  
            dfs(t)  
    }  
}
```

- Note that tree construction and error checking is performed at the same time; SPIN is an **on-the-fly model checker**.
- States are stored in a hash table, and old states are stored on a stack.

Typical Checks

Several checks are typically used to test for properties: deadlock, assertions, invariance, and liveness (LTL):

1. **Sanity check** – random and interactive simulations
2. **Partial check** – use SPIN's bitstate hashing (states are not stored) mode to quickly sweep over the state space.
3. **Exhaustive check** – if bitstate hashing fails, SPIN supports several options to proceed:
 - Compression of state vector
 - Optimization (SPIN options or manual)
 - Abstractions (manual)
 - Bitstate hashing

Invariance

- **Always $P = []P$ where P is a state property:**
 - safety property
 - invariance = global universality or global absence
- Approximately 25% of the properties typically being checked with model checkers are invariance properties, and 48% of the properties are response properties; e.g.:
 - **$[] \text{!flag}$**
 - **$[] \text{mutex} < 2$**
- SPIN supports several ways to check for invariance.

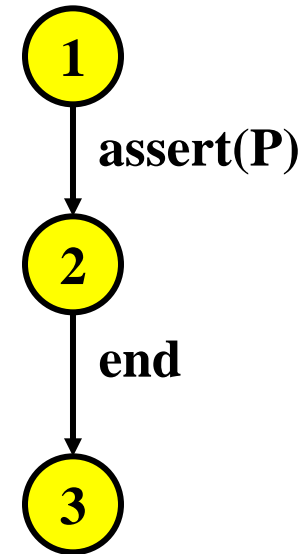
1,2. Monitor process (single assert)

- Proposed in Spin's documentation
- Add the following monitor process to the Promela model:

```
active proctype monitor()  
{  
    assert(P);  
}
```

- Two **variations**:

1. monitor process is created **first**
2. monitor process is created **last**



3. Guarded monitor process

- Drawback of solution “1+2 monitor process” is that the **assert** statement is enabled in every state.

```
active proctype monitor( )  
{  
    assert(P) ;  
}
```



```
active proctype monitor( )  
{  
    atomic {  
        !P -> assert(P);  
    }  
}
```

- The **atomic** statement only becomes executable when P itself is **not** true.

4. Monitor process (do assert)

- From an operational perspective, the following monitor process seems less effective, but there are fewer states:

```
active proctype monitor( )  
{  
    do  
        :: assert(P)  
    od  
}
```



Checking Invariance

- Experimentally, methods 1 and 2 perform the worst -- when checking invariance, these methods should be avoided.
- Method 4 “monitor do assert” performs well, but may change the model if it contains a timeout; e.g., the do-assert loop is always executable, so a timeout will never be executed.
- Overall, method 3 “guarded monitor process” is the most effective and reliable for checking invariance.

Rules of Thumb

(How to construct an efficient Promela model)

■ **Data and variables:**

- ❑ All data ends up in the state vector.
- ❑ More states are generated if a variable can be assigned more values – limit variable size.
- ❑ Limit channel size (e.g., the channel dimension).
- ❑ Prefer local variables over global variables.

■ **Atomicity:**

- ❑ Enclose statements that do not need to be interleaved with atomic or d_step statements.
- ❑ Beware of infinite loops or other semantic changes due to restrictions in interleaving.

■ **Processes:**

- ❑ If possible, combine the behavior of two processes into a single process.

SPIN Summary

■ **Tools:**

- ❑ SPIN – Simple Promela Interpreter
- ❑ XSPIN – SPIN Interface

■ **Observations:**

- ❑ Model checking with SPIN is best at finding errors.
 - ❑ There are many different ways to model the same system in Promela.
 - ❑ Experiment with the Promela models to fine tune the verification model and reduce the search space.
-

Verification

- **Verification** means proving correctness; that is, establishing that a design fulfills certain properties of interest (assertions) or that a particular property will never be satisfied (a never claim).
- **Why is verification needed?**
 - ❑ The proliferation of embedded systems is widespread.
 - ❑ System reliability depends on correct functioning of both hardware and software.
 - ❑ Embedded systems are used in safety-critical control systems in which errors can be fatal or very costly.

Verification versus Testing

- Testing starts with a set of possible test cases, simulates the system on each input, and observes the behavior. In general, testing does not cover all possible executions.
 - On the other hand, verification establishes correctness for **all** possible execution sequences.
-

Techniques for Verification

- **Formal verification:** prove mathematically that the program is correct – this can be difficult for large programs.
 - **Correctness by construction:** follow a well-defined methodology for constructing programs.
 - **Model checking:** enumerate all possible executions and states, and check each state for correctness.
-

Model Checking

- **Problem:** The number of states can be very large.
- **Two Phases:**
 1. **Create a model:** some approximation of the system under construction; e.g., use a finite state model you need to model as well as its environment.
 2. **Verify the model:** determine the properties you want to verify, and check whether the model satisfies the properties.
- The verification exercise is only as good as the model.

Mutual Exclusion

(Incorrect Solution)

```
bit x1 = 0; /* used to indicate that process 1 wants in cs */
bit x2 = 0; /* used to indicate that process 2 wants in cs */
int mutex = 0; /* used to count number of processes in cs */
```

```
proctype P1()
{
    x1 = 1;
    x2 == 0;
    mutex++;
    /* in critical section (cs) */
    mutex--;
    x1 = 0;
}
```


```
proctype P2()
{
    x2 = 1;
    x1 == 0;
    mutex++;
    /* in critical section (cs) */
    mutex--;
    x2 = 0;
}
```


```
proctype monitor()
{
    assert(mutex!=2);
}

init
{
    run P1();
    run P2();
    run monitor();
}
```

Second Attempt

```
/* Second attempt = second.pml */  
bool wantp = false, wantq = false;  
byte critical = 0;
```

```
active proctype p() {  
    do  
        :: !wantq;   
        wantp = true;  
        critical++;  
        assert (critical == 1);  
        critical--;  
        wantp = false;  
    od  
}
```

```
active proctype q() {  
    do  
        :: !wantp;   
        wantq = true;  
        critical++;  
        assert (critical == 1);  
        critical--;  
        wantq = false;  
    od  
}
```

SPIN Output for Second Attempt

pan: assertion violated (critical==1) (at depth 7)

pan: wrote second.pml.trail

(Spin Version 4.3.0 -- 22 June 2007)

Warning: Search not completed

+ Partial Order Reduction

Full statespace search for:

never claim - (none specified)

assertion violations +

cycle checks - (disabled by -DSAFETY)

invalid end states +

State-vector 16 byte, depth reached 11, ... errors: 1 ...

22 states, stored

10 states, matched

32 transitions (= stored+matched)

0 atomic steps

hash conflicts: 0 (resolved)

2.302 memory usage (Mbyte)

Mutual Exclusion (Peterson's Solution)

```
bit x1 = 0; /* used to indicate that process 1 wants in cs */
bit x2 = 0; /* used to indicate that process 2 wants in cs */
int mutex = 0; /* used to count number of processes in cs */
int turn = 0; /* indicates whose turn it is to enter cs */
```

```
proctype P1()
{
    x1 = 1;
    turn = 2;
    (x2 == 0) || (turn == 1);
    mutex++;
    /* in critical section (cs) */
    mutex--;
    x1 = 0;
}
```

```
proctype P2()
{
    x2 = 1;
    turn = 1;
    (x1 == 0) || (turn == 2);
    mutex++;
    /* in critical section (cs) */
    mutex--;
    x2 = 0;
}
```

```
proctype monitor()
{
    assert(mutex!=2);
}
```

```
init
{
    run P1();
    run P2();
    run monitor();
}
```


Mutual Exclusion

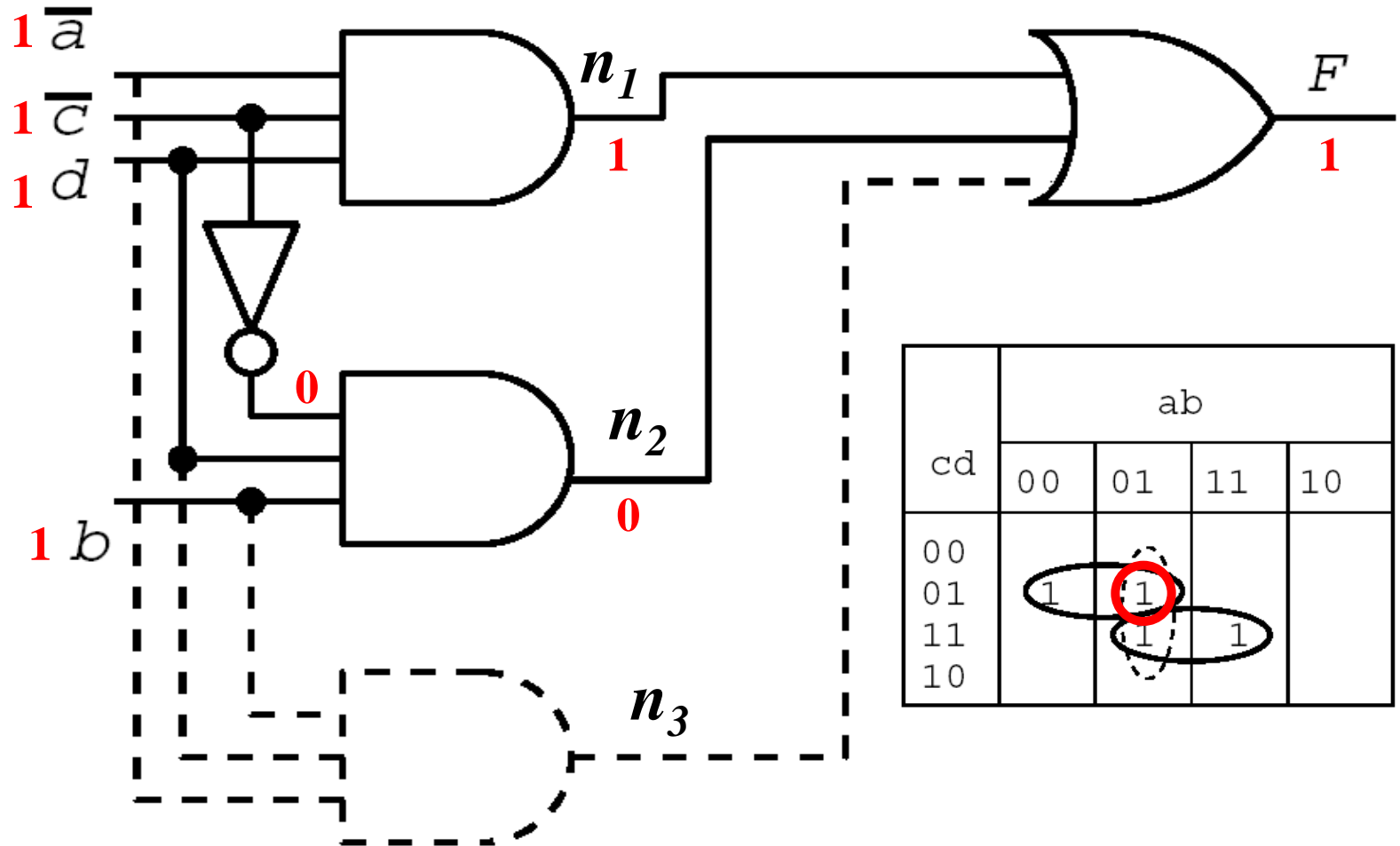
(Peterson's Solution Revised)

```
bool turn, flag[2];
byte ncrit;

active [2] proctype user()
{
    assert(_pid == 0 || _pid == 1);
again:
    flag[_pid] = 1;
    turn = _pid;
    (flag[1 - _pid] == 0 || turn == 1 - _pid);
    ncrit++;
    assert(ncrit == 1); /* critical section */
    ncrit--;
    flag[_pid] = 0;
    goto again
}
```

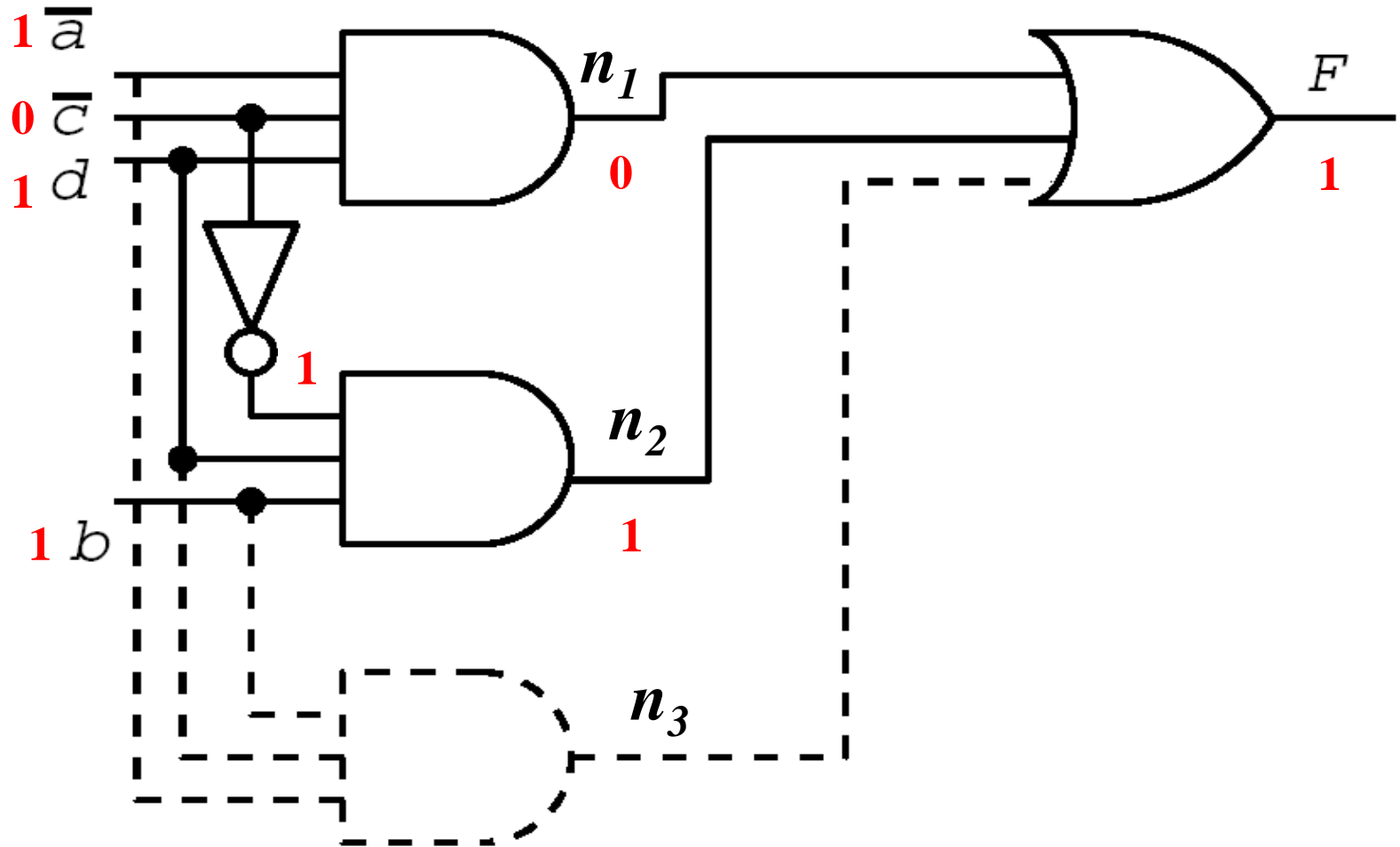
Hardware Circuit Diagram

$$(F = \bar{a}\bar{c}d + bcd)$$



Hazardous Circuit

$$(F = \bar{a}\bar{c}d + bcd)$$



Hardware Gates Model

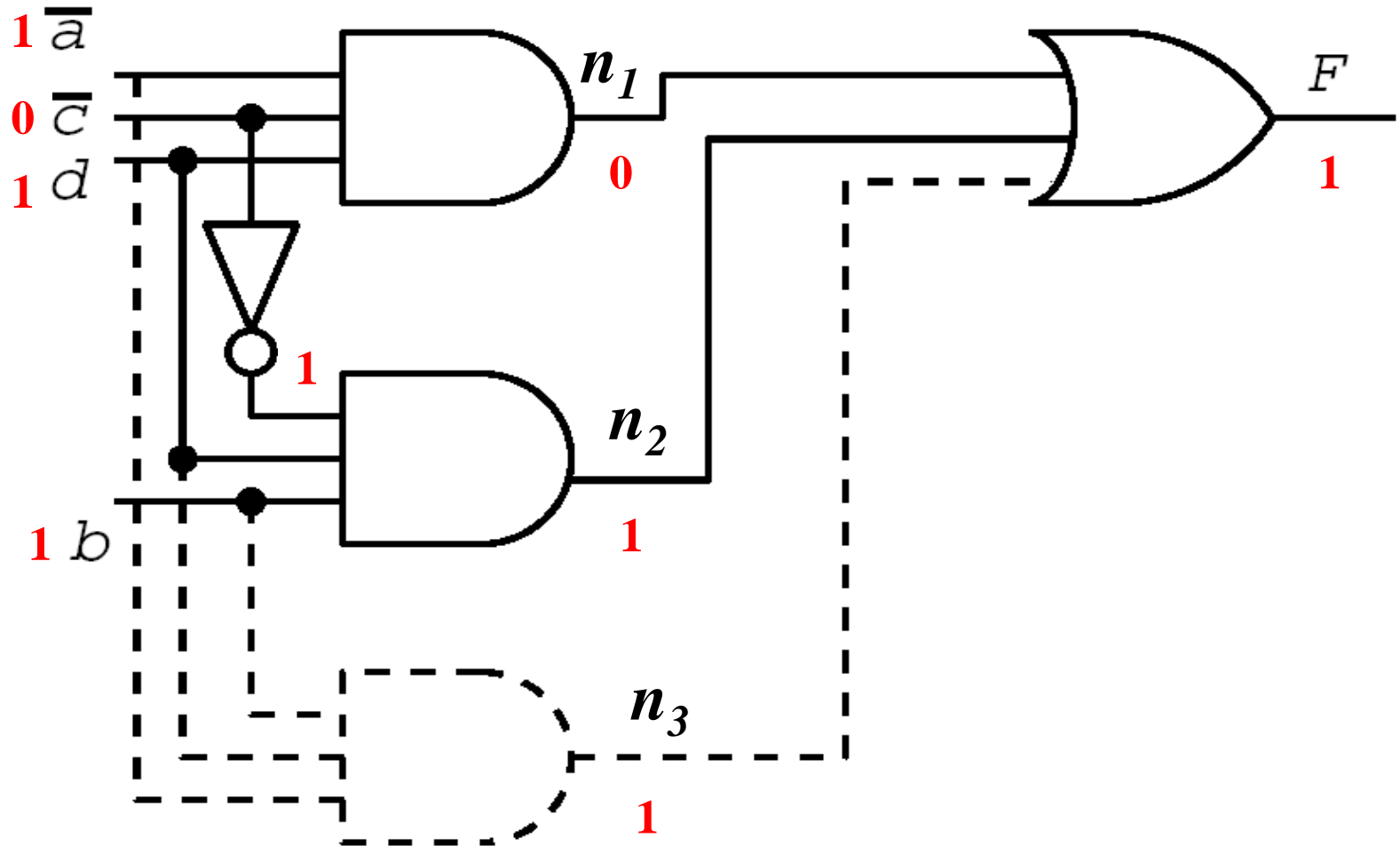
```
bit a, abar, b, c, cbar, d, f;  
bit oldf, n1, n2, newp;
```

```
#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  
}
```

Hazard Removal

$$(F = \bar{a}\bar{c}d + bcd + \bar{a}bd)$$



New Hardware Gates Model

```
bit a, abar, b, c, cbar, d, f;  
bit oldf, n1, n2, n3, 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)  
            :: AND3(abar,b,d,n3)  
            :: OR3(n1,n2,n3,f)  
        fi;  
        newp = 0  
    od  
}
```

Hardware Gates Model (cont.)

```
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() }
}
```

Hardware Gates Model (cont.)

```
never {  
  do  
    :: skip  
    :: (newp == 0 && oldf != f) -> break  
  od;  
  do  
    :: ((newp == 0) && (oldf != f))  
    :: ((newp == 0) && (oldf == f)) -> break  
  od  
}
```

Temporal Logic

- **Temporal logic** is a logical formalism to describe **sequences** of any kind.
 - An **automaton** (or statechart) describes the **actions** of a system.
 - A **temporal logic** formula describes some **property** of the set of sequences.
 - **Basic Paradigm**
 - ❑ System = Automaton
 - ❑ Specification = Temporal Logic Formula
 - ❑ Verification = Satisfaction
-

Linear Temporal Logic (LTL)

- LTL formulae are used to specify temporal properties.
- LTL includes propositional logic and temporal operators:
 - $[]P$ = always P
 - $\langle \rangle P$ = eventually P
 - $P \text{ U } Q$ = P is true until Q becomes true
- **Examples:**
 - Invariance: $[] (p)$
 - Response: $[] ((p) \rightarrow (\langle \rangle (q)))$
 - Precedence: $[] ((p) \rightarrow ((q) \text{ U } (r)))$
 - Objective: $[] ((p) \rightarrow \langle \rangle ((q) \parallel (r)))$

Temporal Claims in SPIN

- The most powerful method for expressing correctness requirements in Promela models is to use a “never” claim.
- Syntax: **never { ... body ... }**
- The body of a never claim expresses behavior that is claimed to be impossible.
- A correctness violation occurs if and only if a temporal claim can be completely matched by a system behavior.
- There can be only one never claim in a model.
- Temporal claims do not specify independent system behavior, they only formalize claims about existing system behavior; e.g., the system behavior does not change when a never claim is added.

Rules for Never Claims

- Every “statement” in a temporal claim must model a proposition with no side-effects; e.g., no assignments, receive, or send statements.
 - Statements do not define system behavior, they are only used to monitor system behavior.
 - To violate a claim, the series of propositions listed in a temporal claim must match the system behavior at every single step of execution.
-

Matching Behavior

■ Example: **never{ P -> Q }**

- ❑ Does this mean that it is never the case that when proposition P becomes true that proposition Q will become true eventually?
- ❑ **No.** Recall that $P \rightarrow Q$ is equivalent to $P;Q$, so Q must become true in the next state ☹.
- ❑ **Solution:**

```
never {  
  S0: do  
    :: P || !P  
    :: P -> break  
  od;  
  S1: do  
    :: !Q  
    :: Q -> break  
  od;  
  S2: skip  
}
```

Inverting a Claim

- Suppose we want to prove that whenever P becomes true, then Q will eventually become true as well; that is, our claim is only violated when the truth of P is **not** followed by the truth of Q.
- This occurs when Q remains false forever, in an infinite cycle, or the execution terminates without Q becoming true.
- In Promela, cyclic behaviors are matched with acceptance labels (more on labels later):

```
never {  
    do  
        :: skip  
        :: P -> break  
    od;  
accept: do  
    :: !Q  
    :: timeout && !Q -> break  
od  
}
```

Validation Labels

- **End State Labels:** In all finite state systems, all execution sequences either terminate or cycle back to some previously visited state.
- The final state in a **terminating sequence** must satisfy the following two criteria:
 - ❑ all processes have reached the end of their code, and
 - ❑ all message channels are empty.

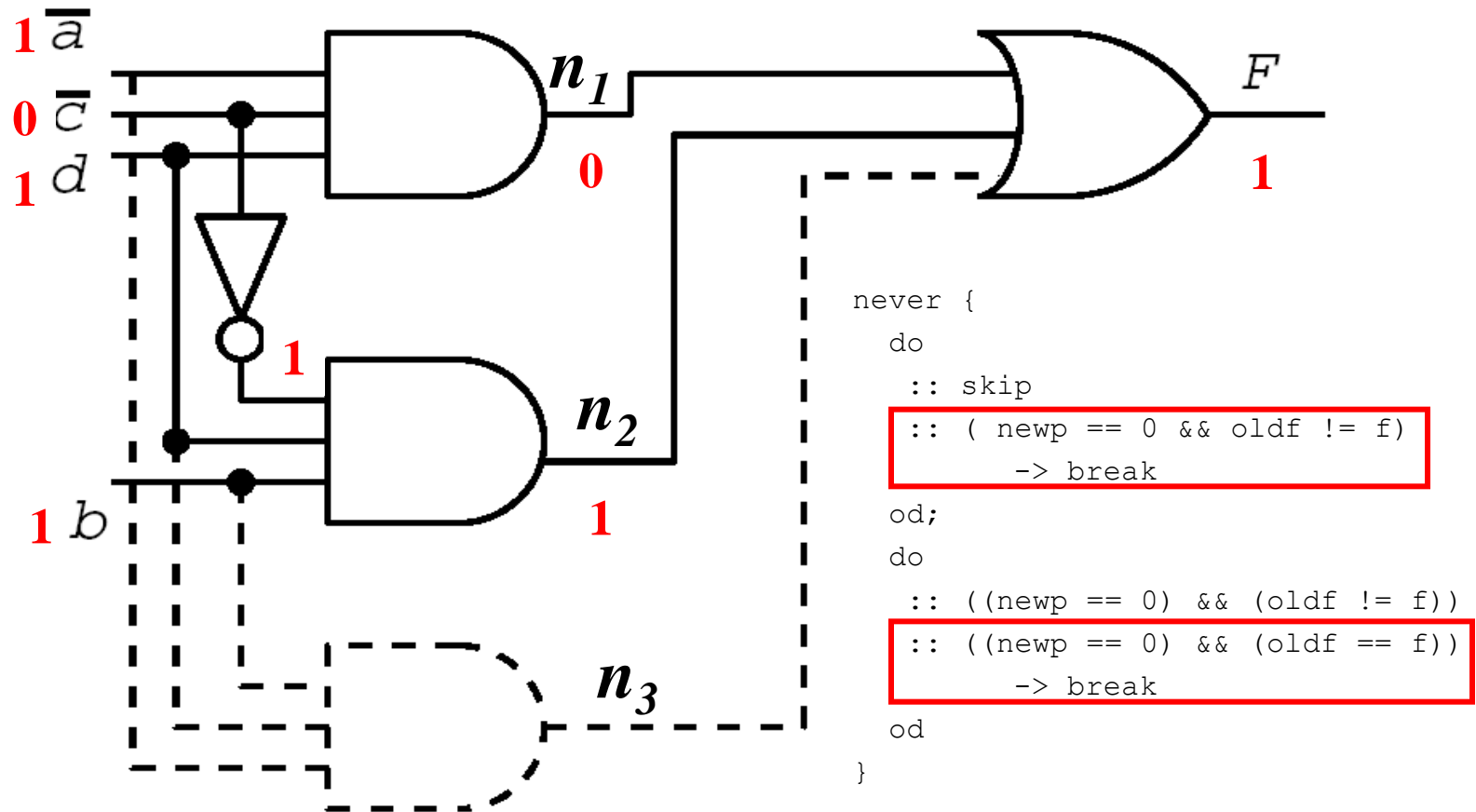
However, not all processes will reach the end of their code and still be in a valid end state; e.g., a server waiting for a connection request. To identify valid end states, use an end state label (**end:** , **endstate:** , etc.).

Validation Labels (cont.)

- **Progress State Labels:** An invalid cyclic execution sequence is a finite sequence of statements that can be repeated infinitely often without making any “progress” in the protocol. The user can specify which statements constitute progress; e.g., incrementing a sequence number, etc., using a progress state label (**progress:** , **progress0:** , **etc.**).
- **Acceptance State Labels:** To express the opposite of a progress condition; e.g., something that cannot happen infinitely often, use acceptance state labels (**accept:** , **acceptance:** , **etc.**). It marks a state that cannot be part of a sequence that can be repeated infinitely often.

Back to the Modelling Problem

$$(F = \overline{a}cd + bcd)$$



Running SPIN

- `spin -a gates2.lta` ← original model
 - `gcc -o pan pan.c`
 - `./pan`
 - → generates trail with violation of never claim
 - `spin -a gates3.lta` ← new model
 - `gcc -o pan pan.c`
 - `./pan`
 - → never claim not violated!
-

Real-Time Promela

```
stmtnt ::=      untimed_stmtnt   |   timed_stmtnt
timed_stmtnt ::= 'when' '{'  $\mu$  '}' untimed_stmtnt
                | 'reset' '{'  $R$  '}' untimed_stmtnt
                | 'when' '{'  $\mu$  '}' 'reset' '{'  $R$  '}' untimed_stmtnt
 $R$  ::= clock ','  $R$ 
 $\mu$  ::= ineq ','  $\mu$ 
ineq ::= clock op int   |   clock op clock '+' int
clock ::=  $x, y, z \in C$  |  $x$  '[' expr ']'
op ::= '<' | '>' | '<=' | '>=' | '=='
```

Here are some examples of timed statements :

```
when{ $x < 4, x \geq 2$ } reset{ $x$ }      B!mymesg ;
when{ $z < 1, y \geq 1$ } reset{ $x, z$ }   a = a*b ;
when{ $x[i] == 1$ }                     goto error ;
```

Example: Timed Mutual Exclusion

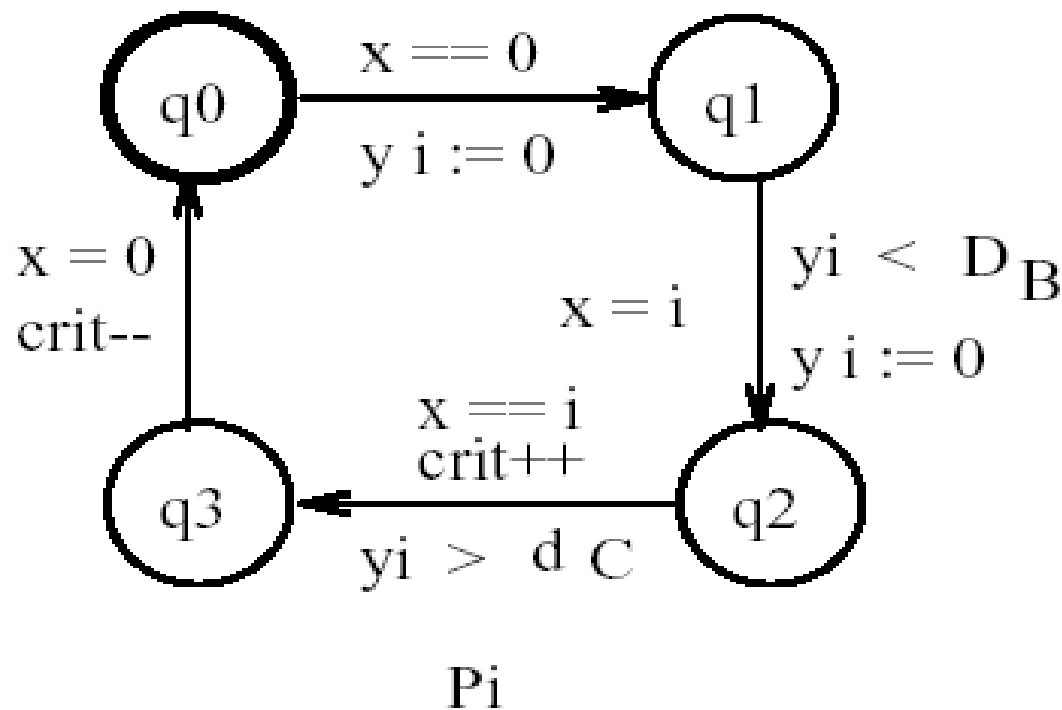


FIGURE 2. Timed mutual exclusion

Timed Mutual Exclusion (cont.)

```
#define N 5 /* number of processes */
#define deltaB 1
#define deltaC 2
#define ErRoR assert(0)
clock y[N];
int x, crit;

proctype P ( byte id )
{
    do ::
        reset{y[id]} x==0 ->
        when{y[id]<deltaB} reset{y[id]} x=id+1 ->
        atomic{ when{y[id]>deltaC} x==id+1; crit++; } ->
        atomic{ x=0; crit--; }
    od
}
```

Timed Mutual Exclusion (cont.)

```
never{  
    skip -> /* to let the processes be activated */  
    do  
        :: crit>1 -> ErRoR  
        :: else  
    od  
}
```

```
init {  
    byte proc;  
    atomic {  
        crit = 0;  
        proc = 1;  
        do  
            :: proc ≤ N ->  
                run P ( proc%N );  
                proc = proc+1  
            :: proc > N -> break  
        od  
    }  
}
```

Example: Train Gate Controller

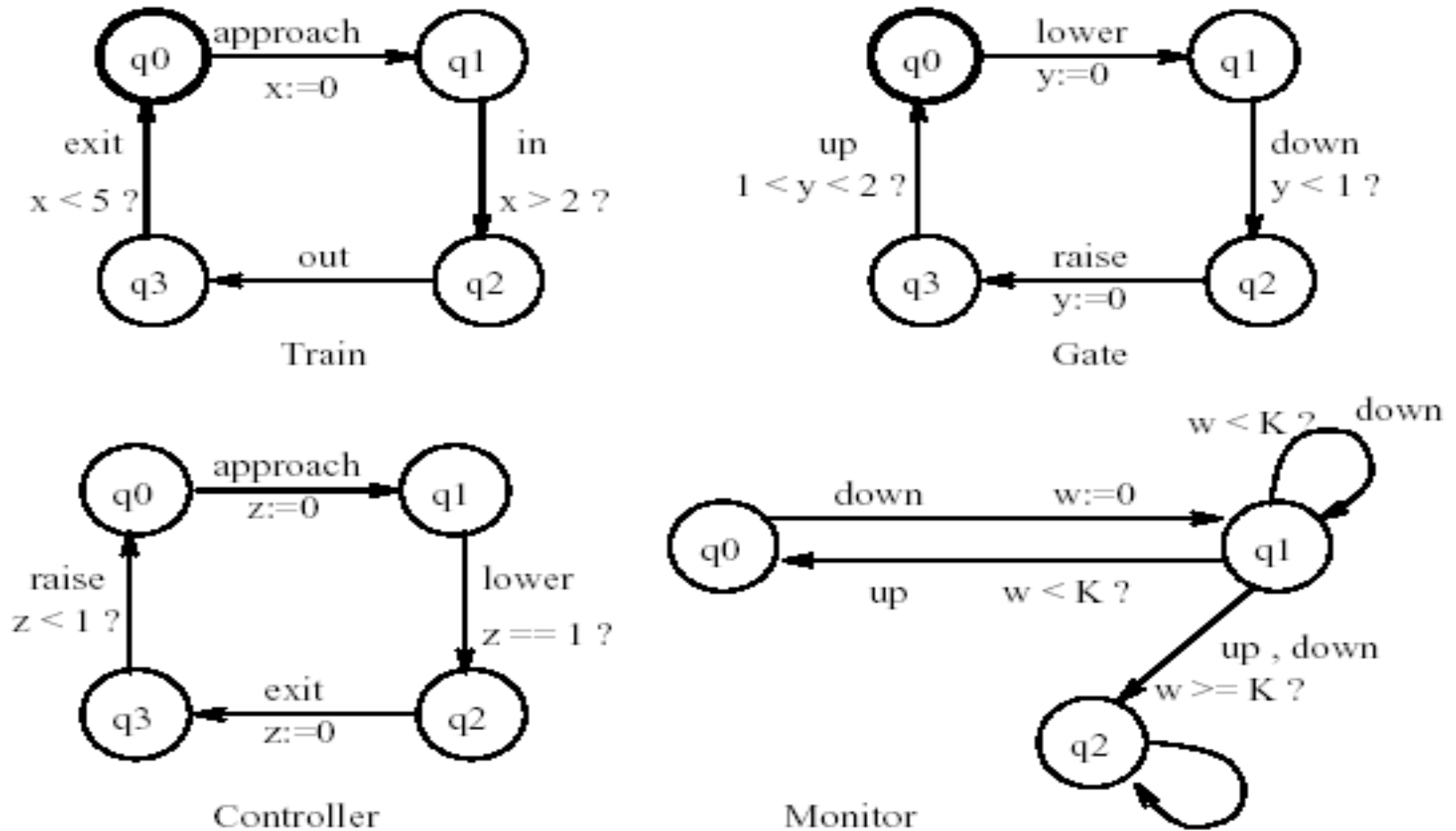


FIGURE 1. Train, Gate, Controller

Summary

- Next Time
 - **Advanced SPIN**