SPIN Model Checker

Abhik Roychoudhury
CS 5219
Department of CS, NUS

The context

- A tool for modeling complex concurrent and distributed systems.
- Provides:
  - Promela, a protocol meta language
  - A model checker
  - A random simulator for system simulation
  - Promela models can be automatically generated from a safe subset of C.

Ideal Usage

- Write programs in C
  - Or C programs for each process in a distributed sys.
- Generate Promela Code from C automatically.
- Use the model checker of SPIN to search through the model represented by the Promela code (automatic verification).
- But …
  - C → Promela tool relatively new.
  - Promela itself is useful for modeling protocols etc.

Our Usage

- Learn Promela, a low-level modeling language.
- Use it to model simple concurrent system protocols and interactions.
- Gain experience in verifying such concurrent software using the SPIN model checker.
- Gives a feel (at a small scale)
  - What are hard-to-find errors ?
  - How to find the bug in the code, once model checking has produced a counter-example ?

Why Promela ?

- Low-level specification language to model finite state sys.
- Models finite state concurrent processes which compute and communicate.
- Fairly extensive coverage of communication
  - Via global shared variables.
  - Via message channels
    - Synchronous communication (hand-shake)
    - Asynchronous communication (buffers)

Why Promela ?

- Extensive support of various control constructs for computation.
  - Assignments, Assert, If, Do
  - Ideas from guarded command languages
- Dynamic creation of processes supported.
  - Gives the flavor of a realistic multi-threaded programming language
  - Yet supported directly by a model checker !!
  - Ideal for our purposes in this course.

Example 0

```
byte state = 0;

proctype A()
{  byte tmp;

     (state==0) -> tmp = state;
      tmp = tmp+1;
      state = tmp;
}

init  { run A() ;  }
```

•**state** : Global Variable

•**tmp** : Local Variable

•**(state==0) -> tmp = state** is a guarded command (blocked if the guard is false).

•Only one process created.

•Final value of **state** is 1

But SPIN allows multiple processes to be created.

Example 1

```
byte state = 0;

proctype A()
{  byte tmp;

     (state==0) -> tmp = state;
     tmp = tmp+1; state = tmp;
}

init  { run A() ; run A();  }
```

What will happen here ?

We need to define how processes are scheduled to determine behaviors.

Process scheduling

- All processes execute concurrently
- Interleaving semantics
  - At each time step, only one of the "active" processes will execute (non-deterministic choice here)
  - A process is active, if it has been created, and its "next" statement is not blocked.
  - Each statement in each process executed atomically.
  - Within the chosen process, if several statements are enabled, one of them executed non-deterministically.
    - We have not seen such an example yet !

Simulation and Verification

- At this point, please note:
  - Promela being an executable specification language, we talk of how Promela programs are executed.
  - Non-determinism broken during simulation.
  - We will mostly use Promela in a different way.
  - Given a Promela program
    - We let SPIN generate a model out of it internally.
    - This model captures ALL possible traces.
    - The model-checker traverses all possible behaviors for debugging (diff. from execution!)

Looking inside a process

- Data Structures
  - Basic types : int, bool, bit, byte
  - Arrays
  - Records (through typedef declarations)
- Just as in C, not much going on here !
- Check SPIN manual for details
  - http://spinroot.com/spin/Man/Manual.html

Statements

- Assignments
- Boolean expressions
  - If true, then no-op else block
- Guarded commands
  - (state == 1) -> tmp = state;
  - Guard and body evaluated separately, be careful !!
  - If you want to evaluate them together
  - atomic { (state == 1) -> tmp = state; }
  - Effect of a test-and-set instruction

Example 1 - Revisited

```
byte state = 0;

proctype A()
{  byte tmp;

     (state==0) -> tmp = state;
     tmp = tmp+1; state = tmp;
}

init  { run A() ; run A();  }
```

Final val. of state can still be 1 ??

Problem of arbitrary shared variable access by several threads.

Verification Example 0

```
bit  flag;
byte sem;
proctype myprocess(bit i)
{   (flag != 1) -> flag = 1;
     sem = sem + 1;
     sem = sem – 1;
     flag =  0;
}
proctype observer() {
       assert( sem != 2 );
}
```

```
init {
     atomic{
          run myprocess(0));
          run myprocess(1));
          run observer();
     }
}
```

All three processes
Instantiated together

Issues

- Initial values of sem, flag not given
  - All possible init. values used for model checking.
- The system being verified is the asynchronous composition myprocess(0) || myprocess(1)
- The property is the invariant
  - G sem ≠ 2
- Local & global invariants can be specified inside code via assert statements.

More on assert

- Of the form assert B
  - B is a boolean expression
  - If B then no-op else abort (with error).
- Can be used inside a process (local invariants)
  - proctype P(…) {   x = … ; assert(x != 2);   …. }
- Or as a separate observer process (global invariants)
  - proctype observer(){ assert(x != 2); }

Warm-up Exercise

- Try out verification example 0 in SPIN
- Try to correct the bug based on the evidence generated by the model checker.

Verification Example 1

```
bit flags[2];
byte sem, turn;
proctype myprocess(bit id)  {
    flags[id] = 1;
    turn = 1 – id;
    flags[1-id] == 0 ||  turn == id;
    sem++;
    sem--;
    flags[id] = 0;
}
```

```
init()  {
     atomic{
          run myprocess(0);
          run myprocess(1);
          run observer();  }
}
proctype observer() {
          assert( sem != 2 );
}
```

Issues

- Can you use SPIN to prove mutual exclusion ?
  - What purpose does turn serve ?
- Arrays have been used in this example.
  - Flags is global, but each element is updated by only one process in the protocol
  - Not enforced by the language features.
- Processes could alternatively be started as:
  - active proctype myprocess(…) {
  - Alternative to dynamic creation via run statement

So far …

- Process creation and interleaving.
- Process communication via shared variables.
- Standard data structures within a process.
- Assignment, Assert, Guards.
- NOW …
  - Guarded IF and DO statements
  - Channel Communication between processes
  - Model checking of LTL properties

Non-deterministic choice

- Choice of statements within a process
  - if
  - :: condition$_1$ -> … ; … ; …
  - …
  - :: condition$_k$ -> … ; … ; …
  - fi;
- If several conditions hold, select and execute any one (more behaviors for verification).
- If none hold, the statement blocks.

Loops

- Similar to the if-fi statement, we have a do-od statement.
- Repeat the choice selection forever.
  - Useful for modeling infinite loops pre-dominant in control software.
- Control can transfer out of the loop via a break statement in the flavor of the C language.

A loop which may terminate

```
byte count;

        proctype counter()
        {
                do
                :: count = count + 1
                :: count = count - 1
                :: (count == 0) -> break
                od;
        }
```

Enumerate the reasons for non-termination in this example

A loop which will not terminate

```
active proctype TrafficLightController() {
     byte color = green;
     do
      :: (color == green) -> color = yellow;
      :: (color == yellow) -> color = red;
      :: (color == red) -> color = green;
      od;
}
```

Channels

- SPIN processes can communicate by exchanging messages across channels
- Channels are typed.
- Any channel is a FIFO buffer.
- Handshakes supported when buffer is null.
- chan ch = [2] of bit;
  - A buffer of length 2, each element is a bit.
- Array of channels also possible.
  - Talking to diff. processes via dedicated channels.

Value-passing

```
chan ch =  [0] of bit;
active proctype sender()                active proctype receiver()
{                                       {  bit  x;
     ch!1;                                  ch?x;
}                                            printf("%d", x);
                                        }
```

The value 1 is passed into local var. x via message passing.
In this example, the message passing was via a handshake
! is output, ? is input

Message retrieval

- ch ? X
  - Retrieve the earliest received (note: FIFO) message from the buffer for ch and store it into the local var. X on the receiver side.
- ch ? 1
  - Same as (ch ? X; X == 1)
- Receiving is always blocked if the corresponding channel buffer is empty.
- Similarly for sending.

An example with channels

```
chan name = [??] of byte;               init { atomic { run A(); run B() } }
proctype A() {
name!124;
name!121;
}

proctype B() {
     byte state;
     name?state
}
```

Enumerate the behaviors when:
?? is 0
?? is 1
?? is > 1

Another (more famous) example

```
#define p 0
#define v 1
chan sema = [0] of { bit };
proctype dijkstra_semaphore() {
byte count = 1;
do
:: (count == 1) -> sema!p; count = 0
:: (count == 0) -> sema?v; count = 1
od
}
```

Another (more famous) example

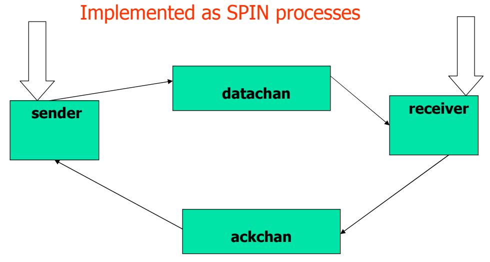
```
proctype user()
{
 do
 :: sema?p;  /* critical section */
    sema!v; /* non-critical section */
od
}

init {
     run dijkstra_semaphore(); run user(); run user(); run user()
}
```

Example: ABP

- Alternating Bit Protocol
  - Reliable channel communication between sender and receiver.
  - Exchanging msg and ack.
  - Channels are lossy
  - Attach a bit with each msg/ack.
  - Proceed with next message if the received bit matches your expectation.
- chan datachan = [2] of { bit };
- chan ackchan = [2] of { bit };

ABP architecture

Implemented as SPIN processes



Sender

```
active proctype Sender()
{    bit out, in;
      do
      :: datachan!out ->
                ackchan?in;
                if
                :: in == out -> out = 1- out;
                :: else fi
       od
}
```

Receiver

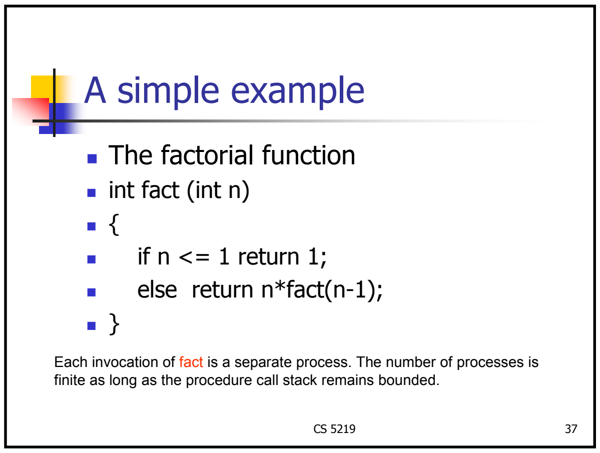
```
active proctype Receiver()
{    bit in ;
      do
      :: datachan?in -> ackchan!in
      :: timeout -> ackchan!in
      od
}
```

Timeouts

- Special feature of the language
  - Time independent feature.
    - Do not specify a time as if you are programming.
  - True if and only if there are no executable statements in any of the currently active processes.
  - True modeling of deadlocks in concurrent systems (and the resultant recovery).

Handling procedures

- Processes in SPIN can be used to model and validate procedures
  - Each instantiation of a procedure (via procedure call) is modeled by spawning of a process.
- How do the called and caller processes communicate ?
  - Common channels
  - Shared Variables
- Makes SPIN useful for software verification.

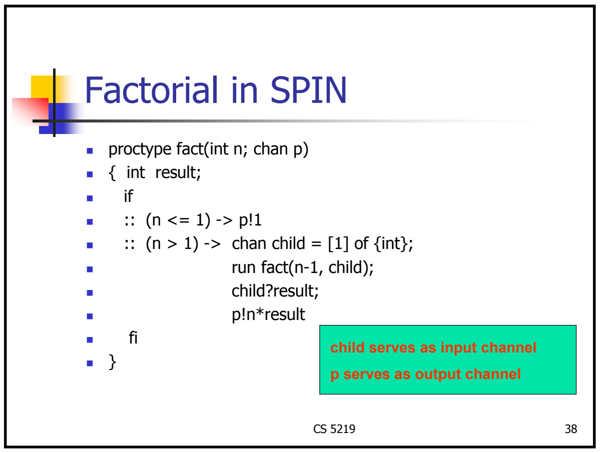


A simple example

- The factorial function
- int fact (int n)
- {
-     if n <= 1 return 1;
-     else return n*fact(n-1);
- }

Each invocation of fact is a separate process. The number of processes is finite as long as the procedure call stack remains bounded.

CS 5219 37



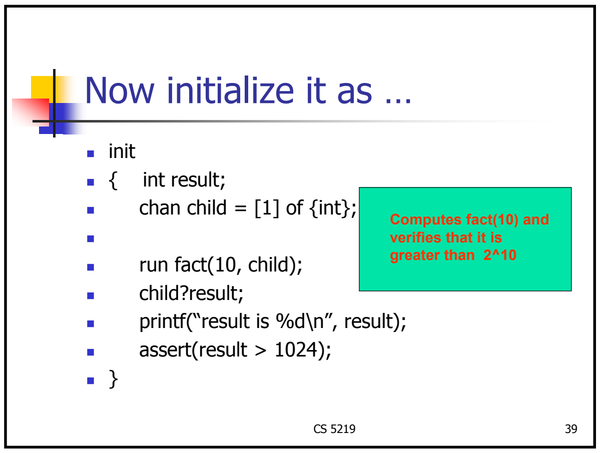
Factorial in SPIN

```
proctype fact(int n; chan p)
{  int  result;
   if
   ::  (n <= 1) -> p!1
   ::  (n > 1) ->  chan child = [1] of {int};
                   run fact(n-1, child);
                   child?result;
                   p!n*result
    fi
}
```

child serves as input channel

p serves as output channel

CS 5219 38

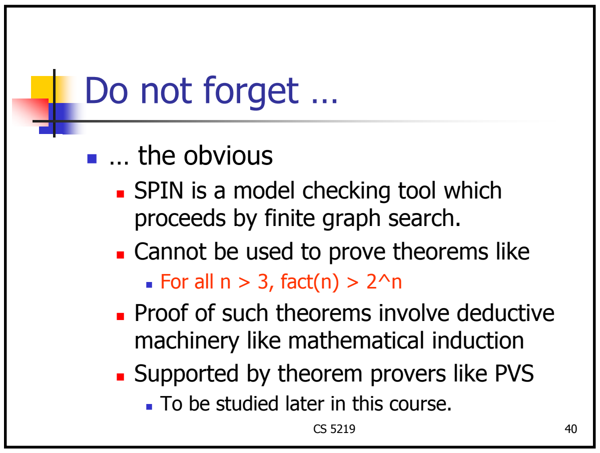


Now initialize it as …

```
init
{    int result;
     chan child = [1] of {int};

     run fact(10, child);
     child?result;
     printf("result is %d\n", result);
     assert(result > 1024);
}
```

Computes fact(10) and verifies that it is greater than 2^10

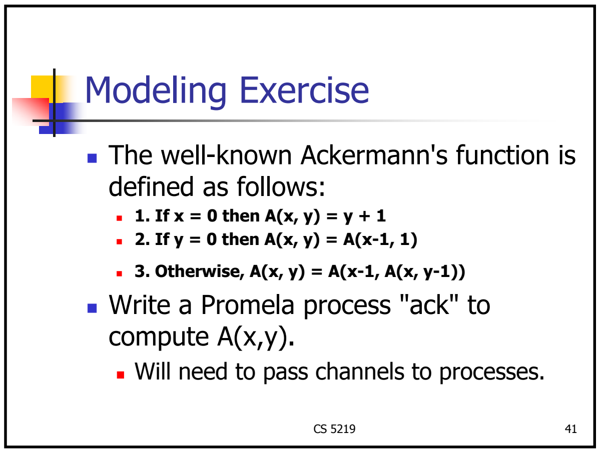
CS 5219 39



Do not forget …

- … the obvious
  - SPIN is a model checking tool which proceeds by finite graph search.
  - Cannot be used to prove theorems like
    - For all n > 3, fact(n) > 2^n
  - Proof of such theorems involve deductive machinery like mathematical induction
  - Supported by theorem provers like PVS
    - To be studied later in this course.

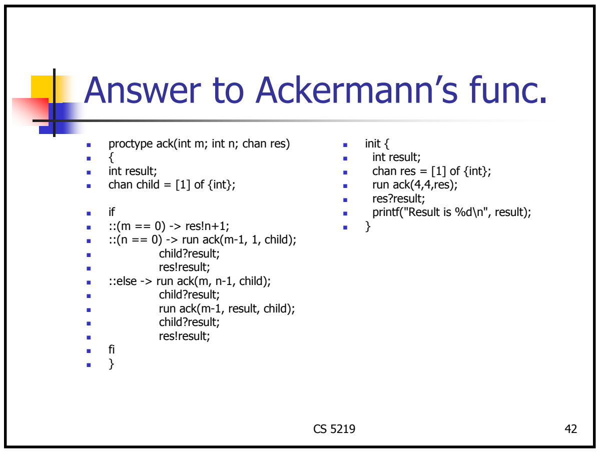
CS 5219 40



Modeling Exercise

- The well-known Ackermann's function is defined as follows:
  - **1. If x = 0 then A(x, y) = y + 1**
  - **2. If y = 0 then A(x, y) = A(x-1, 1)**
  - **3. Otherwise, A(x, y) = A(x-1, A(x, y-1))**
- Write a Promela process "ack" to compute A(x,y).
  - Will need to pass channels to processes.

CS 5219 41



Answer to Ackermann’s func.

```
proctype ack(int m; int n; chan res)
{
int result;
chan child = [1] of {int};

if
::(m == 0) -> res!n+1;
::(n == 0) -> run ack(m-1, 1, child);
              child?result;
              res!result;
::else -> run ack(m, n-1, child);
              child?result;
              run ack(m-1, result, child);
              child?result;
              res!result;
fi
}
```

```
init {
  int result;
  chan res = [1] of {int};
  run ack(4,4,res);
  res?result;
  printf("Result is %d\n", result);
}
```

CS 5219 42

More Modeling Exercises

- Use SPIN to prove mutual exclusion of the semaphore encoding.
- Enough of modeling, let us do some verification.
- Features of PROMELA relevant to verification
  - End, Progress, Accept labels

Part II: Verification using SPIN

Abhik Roychoudhury
CS 5219
Department of CS, NUS

Quotable Quotes

- "I have been fishing all day, I have found a number of fish since the morning, I cannot find any more now, I am pretty sure, there aren't any left!"
  - Folklore
    - Taken from Antonia Bertolino's slides on testing
- Bug finding techniques will ensure worse coverage than fishing in a small pond.

Quotable Quotes

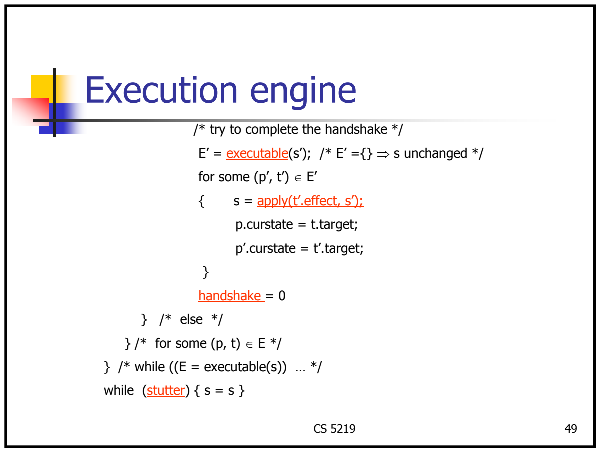
- "If I had eight hours to chop down a tree, I would spend six hours sharpening my axe."
  - U.S. President Abraham Lincoln
    - 1809 – 1865
- Time investment in building verifiers is time well-spent!

Execution engine

- Select an enabled transition of any thread, and execute it.
- A transition corresponds to one statement in a thread.
  - Handshakes must be executed together.
    - *chan x = [0] of {…};*
    - *x!1 || x?data*

Execution engine

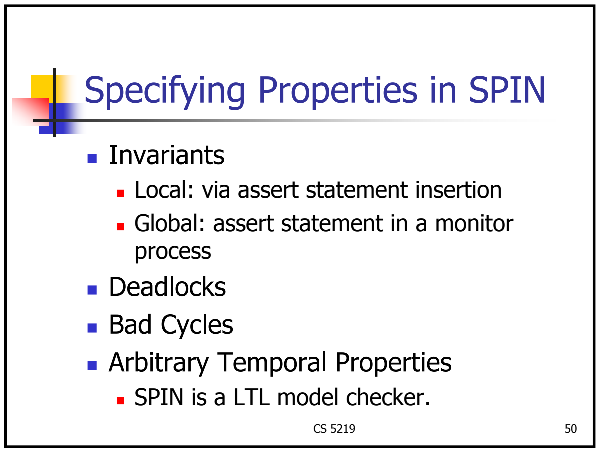
```
while ( (E = executable(s))  != {})
      for some (p,t) ∈ E
      {    s' = apply(t.effect, s);   /* execute the chosen statement */
           if (handshake == 0)
           {        s = s' ;
                    p.curstate = t.target
           }
           else{  …
```



Execution engine

```
                    /* try to complete the handshake */
                     E' = executable(s');  /* E' ={} ⇒ s unchanged */
                     for some (p', t') ∈ E'
                     {       s = apply(t'.effect, s');
                             p.curstate = t.target;
                             p'.curstate = t'.target;
                      }
                     handshake = 0
          }   /*  else  */
       } /*  for some (p, t) ∈ E */
 }  /* while ((E = executable(s))  … */
 while  (stutter) { s = s }
```

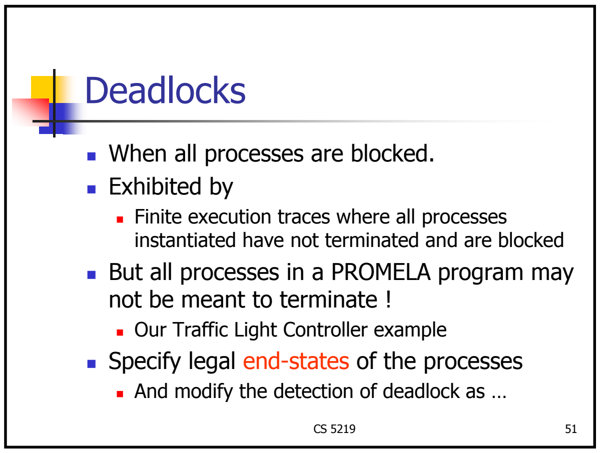
CS 5219 49



Specifying Properties in SPIN

- Invariants
  - Local: via assert statement insertion
  - Global: assert statement in a monitor process
- Deadlocks
- Bad Cycles
- Arbitrary Temporal Properties
  - SPIN is a LTL model checker.

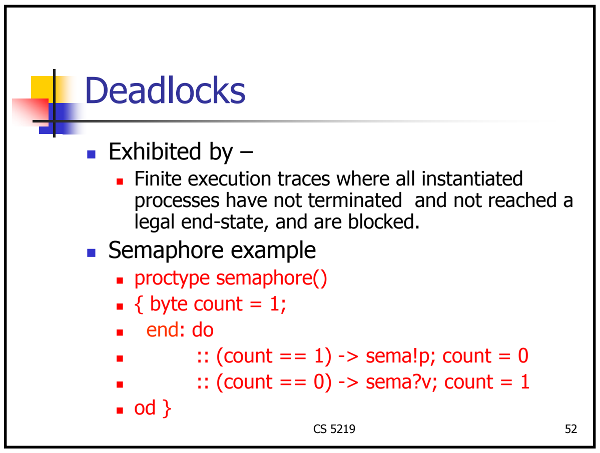
CS 5219 50



Deadlocks

- When all processes are blocked.
- Exhibited by
  - Finite execution traces where all processes instantiated have not terminated and are blocked
- But all processes in a PROMELA program may not be meant to terminate !
  - Our Traffic Light Controller example
- Specify legal end-states of the processes
  - And modify the detection of deadlock as …

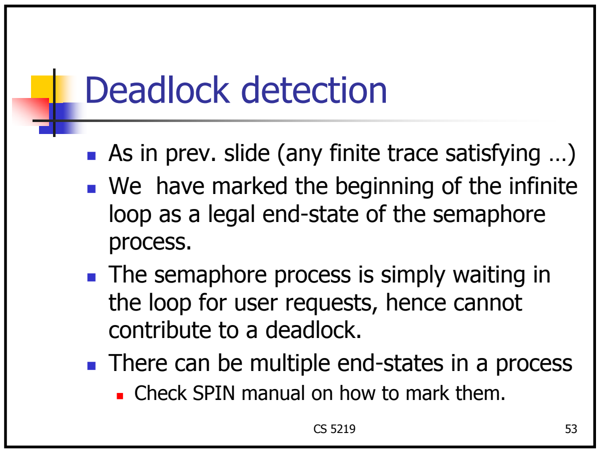
CS 5219 51



Deadlocks

- Exhibited by –
  - Finite execution traces where all instantiated processes have not terminated and not reached a legal end-state, and are blocked.
- Semaphore example
  - proctype semaphore()
  - { byte count = 1;
  -   end: do
  -           :: (count == 1) -> sema!p; count = 0
  -           :: (count == 0) -> sema?v; count = 1
  - od }

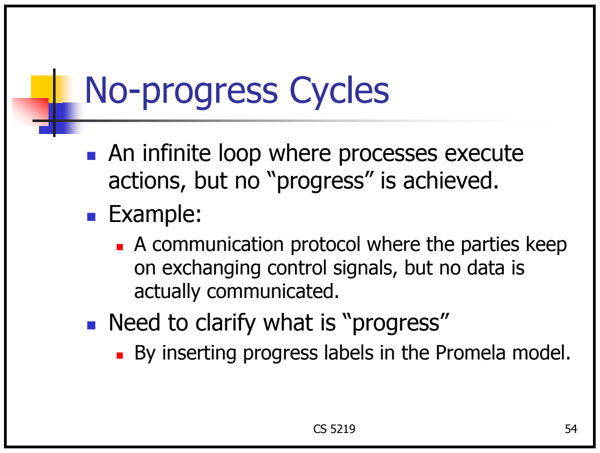
CS 5219 52



Deadlock detection

- As in prev. slide (any finite trace satisfying …)
- We have marked the beginning of the infinite loop as a legal end-state of the semaphore process.
- The semaphore process is simply waiting in the loop for user requests, hence cannot contribute to a deadlock.
- There can be multiple end-states in a process
  - Check SPIN manual on how to mark them.

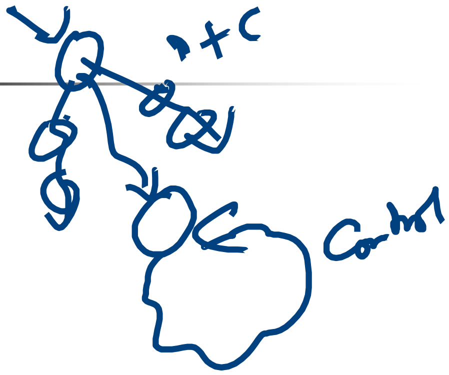
CS 5219 53



No-progress Cycles

- An infinite loop where processes execute actions, but no "progress" is achieved.
- Example:
  - A communication protocol where the parties keep on exchanging control signals, but no data is actually communicated.
- Need to clarify what is "progress"
  - By inserting progress labels in the Promela model.

CS 5219 54



No progress cycles

```
proctype dijkstra()
{  byte count = 1;
   end: do
          :: (count == 1) ->
progress:      sema!p; count = 0
          :: (count == 0) -> sema?v; count = 1
        od
}
```

Verifies that at least one process enters the critical section.

Correctness claims

- Progress labels
  - Any infinite execution cycle contains at least one progress label.
  - No progress cycles are cycles without any progress label
- Acceptance labels
  - No execution trace passes through an accept label infinitely often
  - Model Checking reports an acceptance cycle (if any)
  - Acceptance cycles are cycles with at least one acceptance state.

Acceptance cycles

- A cycle which goes through an "acceptance" state infinitely often
  - A "bad" cycle if the acceptance state is supposed to occur only finitely many times.
  - An acceptance state could mark the state reached after some initialization activity in a protocol.
  - Accept. Cycle $\Rightarrow$ System unintentionally getting reset!
- Can mark acceptance states by "accept" labels in Promela code
  - Labels can be marked by user
  - Accept labels can be automatically generated from user-provided LTL properties to support LTL verification (later !)

Acceptance cycles

```
proctype semaphore()
{  byte count = 1;
   end: do
          :: (count == 1) ->
   progress:   sema!p; count = 0
          :: (count == 0) ->
accept:        sema?v; count = 1
         od
}
```

The acceptance label makes it impossible to loop through P and V operations of the semaphore - this property is false incidentally.

Model Checking

- (P1 || P2 || P3) $\models \varphi$
  - P1, P2, P3 are Promela processes
  - φ is a LTL formula
- Construct a state machine via
  - M, asynchronous composition of processes P1, P2, P3
  - M($\neg\varphi$), representing $\neg\varphi$
- Show that "language" of M $\times$ M($\neg\varphi$) is empty

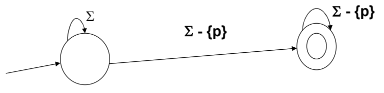
Model Checking

- A given LTL property (rather its negation) is internally represented as an automata.
- This property automata is synchronously composed with the global system automata.
- We then show that the traces accepted by the composition of the system and property automata is empty.
- But the traces are potentially infinite …
  - Finite state automata over infinite inputs

Buchi automata

Mark certain states as acceptance states as usual.

Accept an infinite string if (at least one of) its runs through the automata visits (one or more of) the acceptance states infinitely often. OTHERWISE reject the string.



Accepts infinite strings with finitely many occurrences of p

Buchi automata

- Like conventional finite-state automata
  - $A = (S, \Sigma, I, \rightarrow, F)$
    - S, set of states
    - Σ, a finite alphabet
    - $I \subseteq S$, set of initial states
    - $\rightarrow \subseteq S \times \Sigma \times S$, transition relation
    - $F \subseteq S$, set of final states
  - Notion of acceptance is different

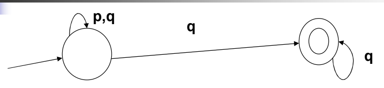
Buchi automata

- Run r of a string $\sigma \in \Sigma^\omega$
  - Sequence of states of A obtained by running σ from an initial state of automata A
  - $r[0] \in I$ and, for all $i \geq 0$, $r[i] \xrightarrow{\sigma[i]} r[i+1]$
- Given a run r,
  - inf(r) = set of states appearing infinitely often in r
  - These are the states that are visited infinitely often on running the infinite string σ
- Language of the automata (notion of acceptance)
  - $L(A) = \{\sigma \mid \sigma \in \Sigma^\omega$ and σ has a run r s.t. $inf(r) \cap F \neq \emptyset \}$

Buchi automata

- Conventional finite state automata over finite strings
  - String accepted if it ends in a final state
- Buchi automata over infinite strings
  - String accepted if it visits at least one final state infinitely often.
    - We need to deal with infinite strings since the system execution traces are infinite.

LTL properties

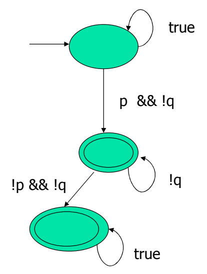


Corresponds to the negation of the LTL property

GF p (assuming $\Sigma = \{p,q\}$)

If the user seeks to verify GFp, SPIN generates Promela code for the negation of the property which will internally construct such an automata.

A more complex property



Represents negation of the LTL property

G (p ⇒ (p U q))

Internally generated by SPIN when the user wants to verify the LTL property.

Acceptance states correspond to accept labels discussed earlier.

Verif. via Acceptance cycles

- Given the Buchi automata for the negation of LTL property (and its acceptance states)
  - SPIN computes a synchronous product of this with the global transition system
  - The property automata should always make a move with the system automata
  - The language of the product automata is non-empty iff it makes the property automata move in a cycle containing acceptance states.
  - Verification achieved by nested depth first search to find such acceptance cycles.

No-progress cycles

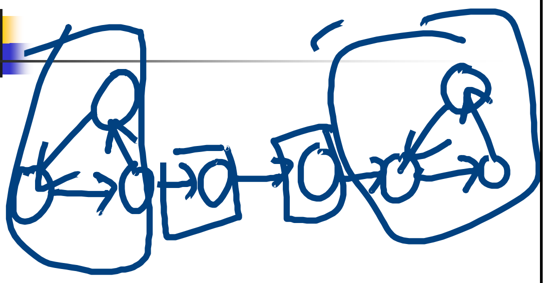
- Absence of no-progress cycles described in LTL as
  - GF progress (verify this !)
- Negation of the property is
  - FG no_progress
    - where no_progress is an atomic proposition which is true in any state where the control location is not marked as progress
- We can compose the program model M with the automata derived from FG no_progress and perform model checking by detecting acceptance cycles.

Important Clarification

- SPIN supports model-checking of arbitrary LTL properties by
  - Converting negation of property
  - Converting negated property to Buchi automata
  - Constructing synchronous product of design's transition system and Buchi automata of negated property
  - Defining accepting states of the Buchi automata to accept labels of the product automata, and
  - Searching for acceptance cycles in the product automata.
- Thus, accept labels are generated automatically from LTL property, and are not directly given by user.

Finding acceptance cycles

- We have reduced LTL model checking to finding acceptance cycles
- How to find acceptance cycles ?
  - One possibility is by SCC detection
    - 1. Compute strongly connected components of the product graph (DFS)
    - 2. Check whether any SCC contains an acceptance state; if yes, an acceptance cycle exists.
  - But …

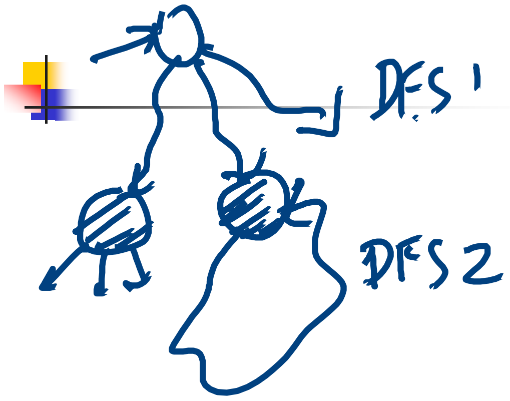


SPIN model checking

- SPIN does not use SCC detection for detecting acceptance cycles (and hence model checking)
- The nested DFS algorithm used in SPIN is more space efficient in practice.
  - SCC detection maintains two integer numbers per node. (*dfs* and *lowlink* numbers)
  - Nested DFS maintains only one integer.
    - This optimization is important due to the huge size of the product graph being traversed on-the-fly by model checker.

Nested DFS in SPIN

- Find acceptance states reachable from initial states (DFS).
- Find all such acceptance states which are reachable from itself (DFS).
- Counter-example evidence (if any) obtained by simply concatenating the two DFS stacks.



Standard DFS on M $\times$ M(φ)

```
procedure dfs(s)
    push s to Stack
    add {s} to States
    for each transition s → s'  do
        if s' ∉ States then dfs(s')
    endfor
    pop s from Stack
end
```

Nested DFS– Step 1

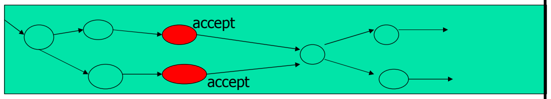
```
procedure dfs1(s)
    push s to Stack1
    add {s} to States1
    if accepting(s) then
        States2 := empty; seed := s; dfs2(s)
    endif
    for each transition s → s' do
        if s'  ∉ States1 then df1(s')
    endfor
    pop s from Stack1
end
```

Nested DFS – Step 2

```
procedure dfs2(s)
    push s to Stack2
    add  {s} to States2
    for each transition s → s' do
        if  s' = seed then report acceptance cycle
                    else  if s' ∉ States2 then df2(s')
        endif
    endfor
    pop s from Stack2
end
```

More on nested DFS

- Space efficient compared to SCC detection
  - Only one id maintained (the DFS number) for each state being visited.
- Time inefficient
  - Different invocations of DFS2 may search the same portion of state space.
  - But DFS2 always starts from scratch.
  - Exercise: how to re-use results of past DFS2 invocations ?



Some Common Questions

- How does the product of the system and property automata work ?
  - How is the interaction between system and property automata achieved ?
- Can we specify LTL properties directly ?
  - Yes, you can do so in SPIN.
- Can we model/verify pgms with procedures
  - Yes.

Connect System and Property

- System model
  - int x = 100;
  - active proctype A()
  - { do
  - :: x %2 -> x = 3*x+1
  - od
  - }
  - active proctype B()
  - { do
  - :: !(x%2) -> x = x/2
  - od
  - }
- Property
  - GF (x = 1)
- Insert into code
  - #define q (x == 1)
- Now try to verify GF q

Enforcing fairness

All LTL counter-examples are acceptance cycles.

No point in reporting acceptance cycles which arise out of unfair scheduling.

If there are K active concurrent processes in the Promela model being verified, all of these processes should have transitions within the acceptance cycle found.

--- Weak fairness requirement

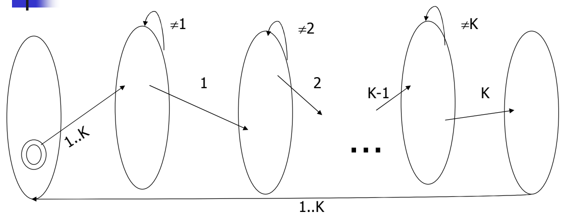
How to enforce this requirement?

Linear blow-up in state space.

Create (K+2) copies of $(P_1||\dots||P_K)\times M(\neg\varphi)$

φ is the property being verified

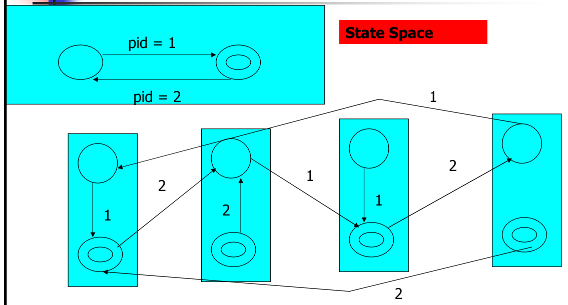
Enforcing fairness



Also, if state s in ith copy has no outgoing transition by P_i add the transition

s in ith copy $\longrightarrow$ s in (i+1)th copy

Enforcing fairness



Let us finish with a real-life situation

- July 4, 1997
  - NASA's Pathfinder landed on Mars.
  - Tremendous engineering feat.
  - Hard to design the control software with concurrency and priority driven scheduling of threads.
  - The SpaceRover would lose contact with earth in unpredictable moments.

The Mars Pathfinder problem

"But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once"." …

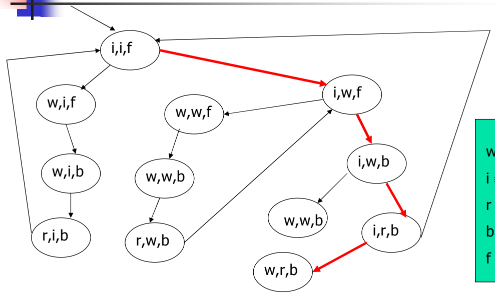
Essence of the problem in SPIN

```
mtype = { free, busy, idle, waiting, running };
mtype H = idle;   mtype L = idle; mtype mutex = free;
```

```
active  proctype high();
{end:  do
        :: H = waiting;
           atomic { mutex == free ->
                    mutex = busy };
           H = running;
          atomic{ H=idle; mutex=free }
         od
}
```

```
active proctype low() provided (H == idle)
{ end: do
        :: L = waiting;
           atomic{ mutex== free->
                   mutex = busy};
            L = running;
            atomic{ L=idle; mutex = free }
     od
}
```

State Space Graph



w =waiting
i =idle
r =running
b =busy
f = free

Source of deadlock

- Counterexample
  - Low priority thread acquires lock
  - High priority thread starts
  - Low priority process cannot be scheduled
  - High priority thread blocked on lock
- Actual error was a bit more complex with three threads of three different priorities
  - Timer went off with such a deadlock resulting in a system reset and loss of transmitted data.

Readings

- http://spinroot.com/spin/Man/Manual.html
  - SPIN manual
- The model checker SPIN (Holzmann)
  - IEEE transactions on software engineering, 23(5), 1997.
- http://spinroot.com/spin/Doc/SpinTutorial.pdf
  - SPIN beginner's tutorial (Theo Ruys)
- Summer school Lecture notes on Software MC
  - See Section 2, Posted under IVLE lesson plan.
- The SPIN model checker: primer and reference manual, by Holzmann (mostly chapters 2,3,7,8)
  - TA168 Hol 2004, RBR in Science Library