INF5140 – Specification and Verification of Parallel Systems

Spring 2017

Institutt for informatikk, Universitetet i Oslo

April 28, 2017

INF5140 – Specification and Verification of Parallel Systems

Lecture 5 - Introduction to Logical Model Checking and Theoretical Foundations

Spring 2017

Institutt for informatikk, Universitetet i Oslo

April 28, 2017

Credits:

- Many slides (all the figures with blue background and few others) were taken from Holzmann's slides on "Logical Model Checking", a course given at Caltech (no longer freely available)
- [Holzmann, 2003, Chapter 2 & 3]

The Spinmodel checker and Promela

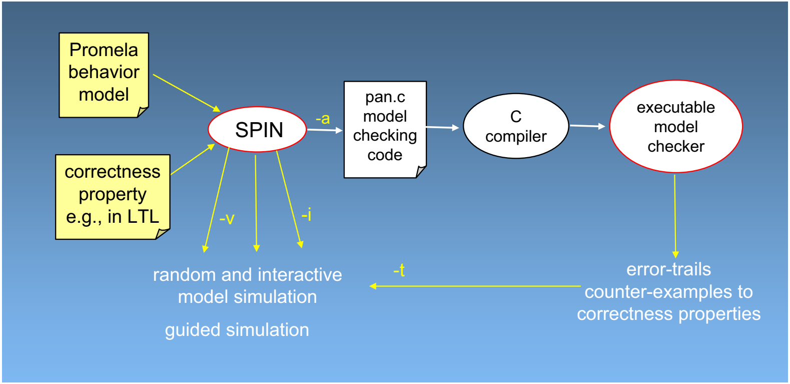
- Spin: "prototypical" *explicit-state LTL* model checker
- Promela: it's input language (for modelling).
- Core: as described theoretically earlier (LTL $\rightarrow$ Büchi).
- many optimizations and implementation "tricks"
  - partial-order reduction
  - various data-flow analyses (dead variables, communication analysis)
  - bitstate hashing (old technique [Morris, 1968])
  - symmetry reduction ...
- repository of material `http://spinroot.com/` (tool, manuals, tutorials, pub's etc)

Spin and Promela

- Promela: PROcess MEta LAnguage
  - system description language/modelling language, **not** a programming lang.
  - emphasis on modeling of process synchronization and coordination, not computation
  - targeted to the description of *software* systems & protocols, rather than hardware circuits
- Spin:[1] Simple Promela INterpreter
  - suppports: *simulation* + verification (i.e., model checking)
  - There are no floating points, no notion of time nor of a clock

[1] It's also the Dutch word for spider ...

Architecture of the tool



The Promela language

Promela

- "input" language for modelling
- C-inspired notation and data structures

Promela features

- asynchronous processes (with shared variables + channel communication)
- buffered and unbuffered message channels
- synchronizing statements
- structured data

```
mtype = { P, C };
mtype turn = P;

active proctype producer()
{
    do
    :: (turn == P) ->
                printf("Produce\n");
                turn = C
    od
}

active proctype consumer()
{
    do
    :: (turn == C) ->
                 printf("Consume\n");
                 turn = P
    od
}
```

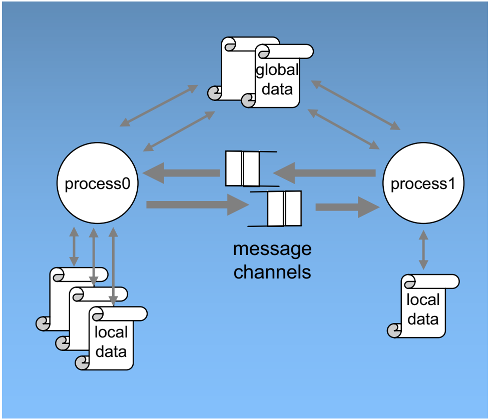
it's a rather trivialized version of P&C

Central concepts

run-time configuration: 3 *basic* ingredients

1. processes
2. global and (process-)local data
3. message channels

focus on *finite state*



Execution model

- remember. LTL model checking based on “finite state automata”
  - (model of) programs seens as FSA/Kripke-structure/transition system[2]
  - Büchi-automata (for checking satisfactin of LTL formulas)

Extended finite state machines

(Often) used for networks of communicating finite state automata, i.e. FSA’s plus FIFO buffers for message passing

- polpular model (indendent from Spin) for procol verification
- for example *LOTOS*
  - international ISO-standard[3]
  - protocol specification language (inspired by algebraic data structures and process algebras,

[2] Assuming that there’s no infinite data types or a stack.

[3] `https://www.iso.org/standard/16258.html`

Scope

Only two levels of scope in Promela

- global
  - global to **all** processes
  - impossible to define variables to a subset of processes
- process local
  - local variables can be referenced from its point of declaration onwards inside the `proctype` body
  - impossible to define local variables restricted to specific blocks

Data types

- C-inspired (for various reasons)
- default initialization to zero[4]
- data types (except channels, which are special)
  - Basic data types
  - records ("structs")
  - 1-dimensional arrays[5]
  - no reals, floats, pointers

[4]Not good practice to rely on uninitialized variables.
[5]At least directly, only 1 dimensional ones are supported.

Basic types

Type	Typical Range	Sample Declaration
bit	0..1	bit turn = 1;
bool	false..true	bool flag = true;
byte	0..255	byte cnt;
chan	1..255	chan q;
mtype	1..255	mtype msg;
pid	0..255	pid p;
short	$-2^{15}..2^{15}-1$	short s = 100;
int	$-2^{31}..2^{31}-1$	int x = 1;
unsigned	$0..2^{n}-1$	unsigned u : 3;

Processes

- basic unit of concurrency
- dynamically creatable with arguments (via `run`) or `active`-keyword
- max 255[6]
- asynchronous "running", no assumption on relative speed, non-deterministic
- interacting via
  - shared variables
  - message passing, with channels.
- basically 3 things one can do with channels (plus some variations)
  - create a channel
  - send to channel
  - receive from channel

[6] But state-space explosion may well kill you before that.

Channel communication

Purpose of channels

1. communication: exchange of data via message passing.[a]
2. synchronization: very generally: reducing possible interleavings (one process has to wait, for instance, wait until a value has been safely received).

[a] An alternative would be shared variable concurrency

- execution of a statement with "synchronization power" enabled or not enabled at a given state
- channels are typed
- sending channel (names) over channels[7]
- no sending of processes over channels

[7] Typing not so "deep" for assuring type correctness of that. So it's not type safe.

Simple channel example

```
chan c = [3] of {chan}  /* global handle, visible to A and B */

active proctype A () {
  chan a;                /* uninitialized local channel */
  c?a                    /* get chan. id from process B */
  a!c                    /* and start using b's channel */
}                        /* dubious typing               */

active proctype B() {
  chan b = [2] of { chan };
  c!b;                /* make channel b available to A     */
  b?c;                /* value of c doesn't really change */
                      /* typewise dubious :-O              */
  0                   /* avoid death of B, otherwise b disappears */
}
```

(Almost) same example in Go

```go
package main
import ("fmt";"time")

var c = make(chan (chan int), 3)

func A() () {
        a := <- c              // receive from c, store in a
        a <- 42                // bounce back a value
}
func B() () {
        var b = make (chan int, 2);
        c <- b
        r := <- b
        fmt.Printf("received:  r = %v\n", r)
}
//------------------------------------------------------------------
func main() {
        go A ();
        go B ();
        time.Sleep(100000)    // while true resp for false{}
}                             //  does not work well.
```

Unlike go: `run`-command in Promela gives back *process id*

Sending and receiving

Sending

```
c!e1,e2,e3
```

enabled only if channel is *not full* (but cf. Spin's `-m` option)

Receiving/retrieving

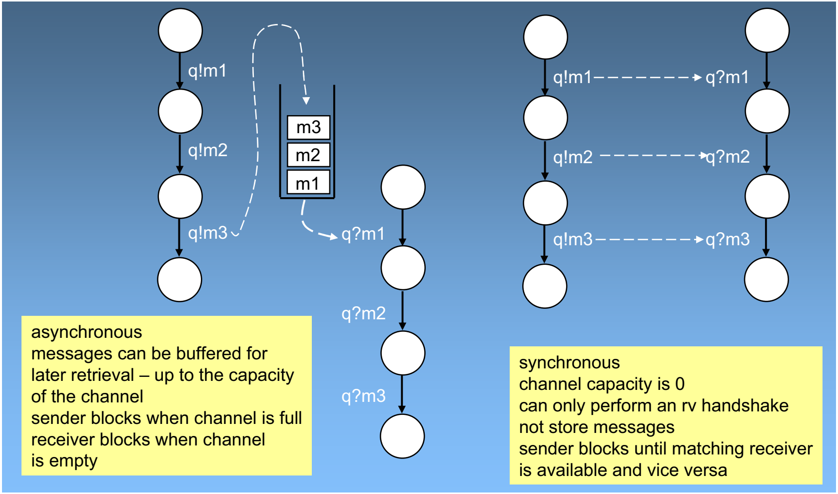
```
c?x1,x2,x3
```

enabled only if channel is *not empty*

- c: "channel"[8], e's: expressions, x's: variables
- special(?) case: channel with capacity = 0: synchronous channel, rendez-vous communication

[8] Variable of appropriate channel type, *referring* to the channel.

Asynchronous vs. synchronous



Receiving can do slightly more fancy things: matching

Matching with constant

If some of the parameters of the receive op ? is a constant (instead of variable) ⇒ receive executable only if the constant parameter(s= match the values of the corresponding fields in the message to be received.

- Note: receiving is a side-effect operation, as in Hoare's CSP, ≠ in Milner's CCS
- eval for matching on (current) content of a variable

```
c?eval(x1),x2,x3
```

Variants

- Sorted send: q!!n,m,p
  - Like q!n,m,p but adds the message n,m,p to q in numerical order (rather than in FIFO order)
- Random receive: q??n,m,p
  - Like q?n,m,p but can match any message in q (it need not be the first message)
- "Brackets": q?[n,m,p]
  - It is a side-effect free Boolean expression
  - It evaluates to true precisely when q?n,m,p is executable, but has no effect on n,m,p and does not change the contents of q
- "Braces": q?n(m,p)
  - Alternative notation for standard receive; same as q?n,m,p
  - Sometimes useful for separating type from arguments
- Channel polls: q?<n,m,p>
  - It is executable iff q?n,m,p is executable; has the same effect on n,m,p as q?n,m,p, but does not change the contents of q

Food for thought

- send and receive: sync. statements!, receive with side effects on variables
- Known knowns:
  - send and receive: **not** expressions, but *i/o statements* (see also 2 slides later)
    - `(a>b && qname?msg0)` illegal
    - `(a>b && qname?[msg0])` fine (or at least legal).
      Expression `qname?[msg0]` is *true* when `qname?msg0` would be executed at this point (but the actual receive is not executed)
- known unknowns: what happens for
  - `c?x1,x2` if the *x*s are global vars with a race condition
  - is the receive at least atomic (and what's `c?x,x`),
  - what about `c?x,eval(x)`
    - does the second one refer to the value of *x* *before* the receive
    - or: does it guarantee that 2 equal values are sent (left-to-right)?
  - similar headaches for send? and the other variants?
  - keep an eye also on "select"-statements!
- the pragmatist's advice: don't program/model like that

Execution

- concurrency $\Rightarrow$ need for synchronization
- depending on the system state each statement
  - executable (aka: enabled)
  - blocked (aka: not enabled)
- cf. also the concept of *guarded commands*
- Promela looks often like C, but that may be deceiving, in particular:

  > *expressions (which have no side-effects!) are executable if they eval. to true or a non-zery integer value*

- cf.

```
(a==b);
```

6 commandmends for executability of basic statements

Unconditionally enabled

- assignment: `x++, x--, x = x+1, x = run P()`
  - `b = c++` is not a valid expression (right-hand side is not side-effect free)
- print: `printf(‘‘x = %d\n’’, x)`
- assertion: `assert(1+1==2)`

Conditionally enabled

- expression statement: when true/non-zero[a]

  `(x), (1), run P(), skip, true, else, timeout`

- channel ops: Executable
  when target channel is non-full resp. non-empty (and matching)

  `q!ack(m)`   `q?ack(n)`

[a] `else` is weird: predefined variable

5 groups of compound control flow

- Basic statements (so far)
  - print, assignment, assertions, expressions, send and receive
  - Notice that `run` is not a statement but an operator and `skip` is an expression (equivalent to `(1)` or *true*)
- Five ways to define control flow
  1. Semicolons + gotos and labels
  2. structuring aids (or hacks)
     - inlines
     - macros
  3. atomic sequences (indivisible sequences)
     - `atomic {...}`
     - `d_step {...}`
  4. Non-deterministic selection and iteration
     - `if ... fi`
     - `do ... od`
  5. Escape sequences (for error handling/interruptions)
     - `{...} unless {...}`

Selection

The (non-deterministic) `if` statement is inspired on Dijkstra's guarded command language

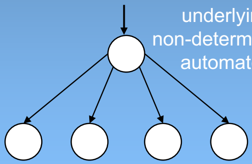
```
/* find the max of x and y */
if
:: x >= y -> m = x
:: x <= y -> m = y
fi
```

```
/* pick a number 0..3 */
if
:: n=0
:: n=1
:: n=2
:: n=3
fi
```

non-deterministically assigns a value to **n** in the range 0..3

```
if
:: (n % 2 != 0)  -> n = 1
:: (n >= 0)      -> n = n-2
:: (n % 3 == 0)  -> n = 3
:: else /* -> skip */
fi
```

the else guard is executable iff *none* of the other guards is executable.



Selection

`else` is a predefined variable

where in C one writes:

```
if (x <= y)
        x = y-x;
y++;
```

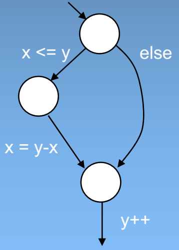
i.e., omitting the 'else'

in Promela this is written:

```
if
:: (x <= y) -> x = y-x
:: else
fi;
y++
```

i.e., the 'else' part cannot be omitted

no need to add "-> skip"



in this case 'else' evaluates to:
!(x <= y)

the else clause always has to
be explicitly present
without it, the if- statement would
block until (x<=y) becomes true
(it then gives only *one* option for behavior)

Selection

`timeout` is also a predefined variable

```
if
:: q?msg -> ...
:: q?ack -> ...
:: q?err -> ...
:: timeout -> ...
fi
```

checking for bad timeouts:
spin –Dtimeout=true model

wait until an expected message arrives, or recover when the system as a whole gets stuck (e.g., due to message loss)

note carefully that using‘else’ intead of ‘timeout’ is dubious in this context

- `else` and `timeout` are related
  - both predefined Boolean variables
  - their values are set to *true* or *false* by the system, depending on the context
- They are, however, not interchangeable
  - `else` is true iff no other statement in the *process* is executable
  - `timeout` is true iff no other statement in the *system* is executable
- A `timeout` may be seen as a system level `else`
- Are these equivalent?

```
if
:: q?msg -> ...
:: q?ack -> ...
:: timeout -> ...
fi
```

```
if
:: q?msg -> ...
:: q?ack -> ...
:: else -> ...
fi
```

- No! In the second, if a message is not received when the control is at the *if* then the *else* is taken immediately

Repetition

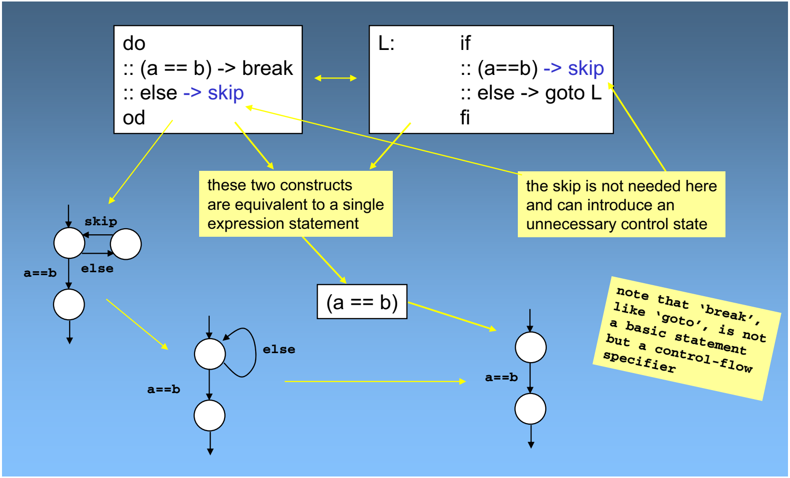
The `do` statement is an `if` statement caught in a cycle

```
do
:: guard1 -> stmnt1.1; stmnt1.2; stmnt1.3;...
:: guard2 -> stmnt2.1; stmnt2.2; stmnt2.3;...
::...
:: guardn -> stmntn.1; stmntn.2; stmntn.3;...
od
```

- Only a `break` or a `goto` can exit from a `do`
- A `break` transfers control to the end of the loop

Repetition

There are many ways of writing a waiting loop, by exploiting the executability rules it's possible to simplify the model



State space explosion, interleaving, and synchronization

- explicit state model checking
- *non-deterministic scheduling*, the only way restriction is "synchronization"
- more interleaving/more scheduling or suspension points: *larger* state-space
- synchronization: for "programming" correctly
- more *coarse-grained* parallelism: smaller state-space
- Cf: *ACID*
- two forms: `atomic` and `d-steps`

Atomic Sequences

```
atomic { guard -> stmnt1; stmnt2; ... stmntn }
```

- executable if the guard statement is executable
- any statement can serve as the guard statement
- executes all statements in the sequence *without* interleaving with statements in other processes
- if any statement other than the guard blocks, atomicity is lost atomicity can be regained when the statement becomes executable
- example: mutual exclusion with an indivisible test&set:

```
active [10] proctype P()
{ atomic { (busy == false) -> busy = true };
  mutex++;

  assert(mutex==1);

  mutex--;
  busy = false;
}
```

Deterministic Steps

d_steps are more restrictive and more efficient than atomic sequences

```
d_step { guard -> stmnt1; stmnt2; ... stmntn }
```

– like an atomic, but *must be deterministic* and *may not block* anywhere

– especially useful to perform intermediate computations with a deterministic result, in a single indivisible step

```
d_step { /* reset array elements to 0 */
        i = 0;
        do
        :: i < N -> x[i] = 0; i++
        :: else -> break
        od;
        i = 0
      }
```

– atomic and d_step sequences are often used as a model reduction method, to lower complexity of large models (improving tractability)

Atomic Sequences, Deterministic Steps and Gotos

- goto-jumps into and out of atomic sequences are allowed
  - atomicity is preserved only if the jump starts inside on atomic sequence and ends inside another atomic sequence, and the target statement is executable
- goto-jumps into and out of d_step sequences are forbidden

```
d_step {
          i = 0;
          do
          :: i < N -> x[i] = 0; i++
          :: else -> break
          od
};
x[0] = x[1] + x[2];
```

this is a jump out of the d_step sequence and it will trigger an error from Spin

the problem is prevented in this case by adding a "; skip" after the od keyword – there's no runtime penalty for this, since it's inside the d_step

Deterministic Steps vs Atomic Sequences

- Both sequences are executable only when the first (guard) statement is executable
  - `atomic`: if any other statement blocks, atomicity is lost at that point; it can be regained once the statement becomes executable later
  - `d_step`: it is an error if any statement other than the (first) guard statement blocks
- Other differences:
  - `d_step`: the entire sequence is executed as one single transition
  - `atomic`: the sequence is executed step-by-step, but without interleaving, it can make non-deterministic choices

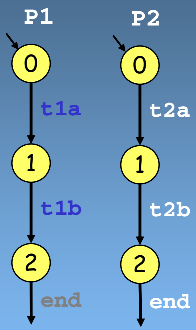
Remarks

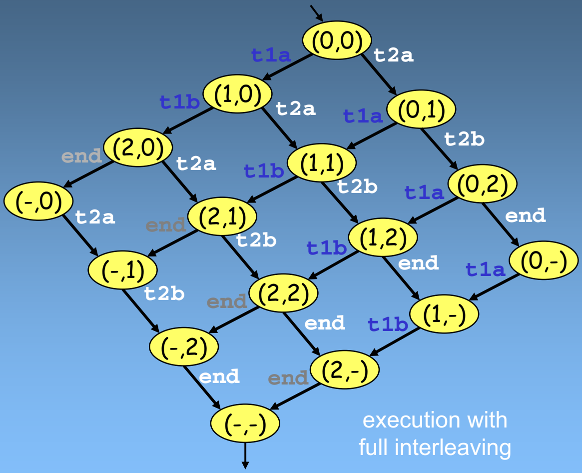
- Infinite loops inside atomic or `d_step` sequences are not detected
- The execution of this type of sequence models an indivisible step, which means that it cannot be infinite

Deterministic Steps and Atomic Sequences

```
active proctype P1() { t1a; t1b }
active proctype P2() { t2a; t2b }
```

execution
without atomics or d_steps



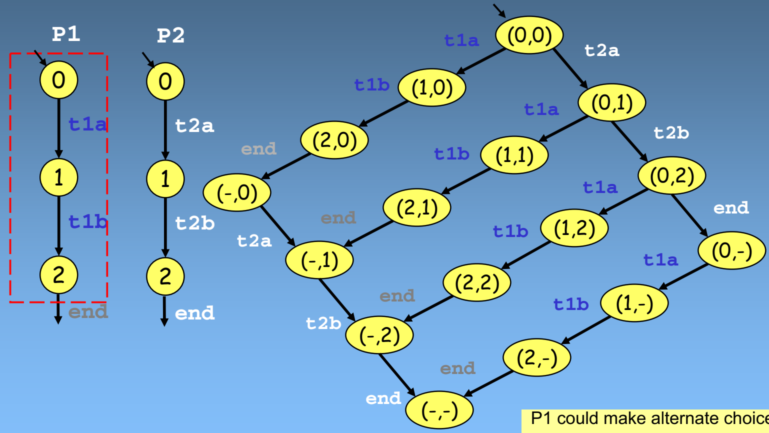


Deterministic Steps and Atomic Sequences

```
active proctype P1() { atomic { t1a; t1b } }
active proctype P2() { t2a; t2b }
```

execution with one atomic sequence

P2 can be interrupted, but not P1



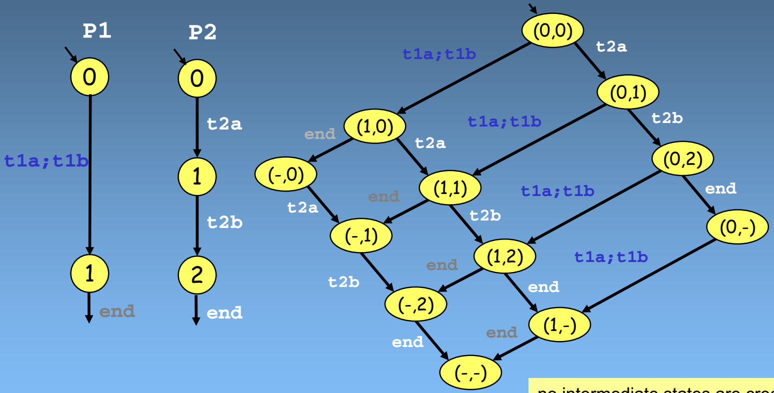
P1 could make alternate choices at the intermediate states (e.g., in if or do-statements)

Deterministic Steps and Atomic Sequences

```
active proctype P1() { d_step {t1a; t1b} }
active proctype P2() { t2a; t2b }
```

execution with a d_step sequence

P1 now has only one transition…



no intermediate states are created: faster, smaller graph, but no non-determinism possible inside d_step sequence itself

Escape sequences

- Syntax: `{ P } unless { Q }`
- Execution starts with the statements from `P`
- Before executing each statement in `P`, the executability of the first statement in `Q` is checked
- Execution of `P` statements continue only if the first instruction of `Q` is not executable
- As soon as the `Q` first statement can be executed, then control changes and execution continues in `Q`
- Example

```
A; { do
     :: b1 -> B1
     :: b1 -> B1
     ...
     od }  unless { c -> C };
D
```

`c` acts here as a *watchdog*: as soon as it becomes true, `C` is executed and then `D`

Inline definitions

- somewhere in between a macro and a procedure
- used as replacement text with *textual* name substitution through parameters (it is a named piece of text with optional parameters)
- an inline is *not* a function – it cannot return values to the caller
- can help to structure a model
- compare:

```
#define swap(a,b)  tmp = a; \
                   a = b; \
                   b = tmp
```

```
inline swap(a,b) {
        tmp = a;
        a = b;
        b = tmp
}
```

looks a little cleaner
line nr refs are better

hint:
when confused, use
spin –I spec.pml
to show the result of all inlining
and macro preprocessing operations...

Specification & claims

Model checking: specifying (desired) behavior

model checking $P \models^? \varphi$:

- spec. what the program *does*
- spec. what the program *should (not) do*

- Side remark:
  - remember: model of the system is not (mostly) the program/system itself
  - One can also interpret the *model* as description of the "desired" system behavior, use it for monitoring etc.

The theoretician's view

Program models are Kripke-structures and specifications are LTL formulas (which can be translated to Büchi-automata). Build the joint transition system and check (iterated) reachability. Problem solved, next question ...

Pragmatics

- Promela: “user-friendly” modelling language (with a Kripke-semantics)
  - “programming” in Promela models/describes program behavior
  - the Spin execution engines executes the model (simulation or state exploration)

Separating desired from undesired behavior

Similar to the fact that naked Kripke structures may not be ideal for easy modelling, Spin offers (besides LTL) pragmatically useful ways to specify (un)-desired behavior

A Spin model consists of

- behavior specification (*what is possible*)
  - Asynchronous process behavior
  - Variables, data types
  - Message channels
- logical correctness properties (*what is valid*)
  - assertions
  - end-state, progress-state, and acceptance state labels
  - never claims
  - trace assertions
  - temporal logic formulae
  - default properties checked automatically:
    - absence of system deadlock
    - absence of dead code (unreachable code)

Basic assertions

basic assertion

```
assert(expression)
```

- most straightforward form of "specification"
- often pragmatically: sprinkle the model/program code with "logical" variables + add assertions

```
byte state = 1;
active proctype A()
{         (state == 1) -> state++;
          assert(state == 2)
}
active proctype B()
{         (state == 1) -> state--;
          assert(state == 0)
}
```

Beware of (non-)atomicity

```
byte state = 1;
active proctype A()
{          (state == 1) -> state++;
           assert(state == 2)
}
active proctype B()
{          (state == 1) -> state--;
           assert(state == 0)
}
```

```
$ spin -a simple.pml
$ gcc -o pan pan.c
$ ./pan -E          # -E means ignore invalid endstate errors...
pan: assertion violated (state==2) (at depth 6)
pan: wrote simple.pml.trail
...
```

```
$ spin -t -p simple.pml
  1:    proc  1 (B) line   7 "simple.pml" (state 1) [((state==1))]
  2:    proc  0 (A) line   3 "simple.pml" (state 1) [((state==1))]
  3:    proc  1 (B) line   7 "simple.pml" (state 2) [state--]
  4:    proc  1 (B) line   8 "simple.pml" (state 3) [assert((state==0))]
  5:    proc  0 (A) line   3 "simple.pml" (state 2) [state++]
spin: line   4 "simple.pml", Error: assertion violated
spin: text of failed assertion: assert((state==2))
```

Preventing the Race

```
byte state = 1;
active proctype A()
{         atomic { (state == 1) -> state++ };
          assert(state == 2)
}
active proctype B()
{         atomic { (state == 1) -> state-- };
          assert(state == 0)
}
```

we added two atomic sequences to create indivisible test&sets

```
$ spin -a simple.pml
$ gcc -o pan pan.c
$ ./pan -E            # -E means ignore invalid endstates...
(Spin Version 4.1.0 -- 6 December 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      - (disabled by -E flag)

State-vector 20 byte, depth reached 3, errors: 0
       6 states, stored
       0 states, matched
       6 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

unreached in proctype A
        (0 of 5 states)
unreached in proctype B
        (0 of 5 states)
```

Q: are there invalid endstates?

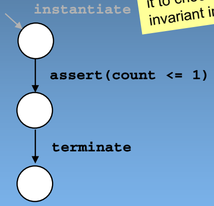
nothing is unreachable

System invariants using basic assertions

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
    do
    :: sem!p ->
       sem?v
    od
}

active [5] proctype user()
{
    do
    :: sem?p ->
           count++;
           /* critical section */
           count--;
       sem!v
    od
}
```

```
active proctype invariant()
{
    assert(count <= 1)
}
```

Q: how expensive is it to check the invariant in this way?

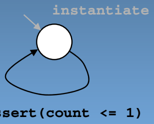


adding active proctype invariant multiplies the search space 3x... (from 16 reachable states to 48)

A small (but easy) improvement

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
   do
   :: sem!p ->
      sem?v
   od
}

active [5] proctype user()
{
   do
   :: sem?p;
         count++;
         /* critical section */
         count--;
      sem!v
   od
}
```

```
active proctype invariant()
{
   do :: assert(count <= 1) od
}
```



no increase in number of reachable states (more transitions, but not more states)

can also put the assertion inside proctype user to check it only when the value of the expression could change

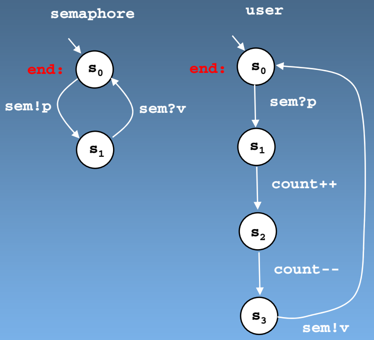
End states

- for checking *deadlock* states: distinguish valid system `end` states from invalid ones
- default: valid end states = end-of-code for all processes
- Not all the processes, however, are meant to reach the end of its code (e.g., waiting loop or state)
- special labels to tell the verifier that those states are valid end states: end-state labels
- label with 3-letter prefix `end`
  - Examples: `endone`, `end_two`, `end_whatever_you_want`
- one of 3 meta labels
- Spin checks invalid end states by default[9]

[9] It is possible to disable it by calling Spin with the `E+` option.

Example: Mutex & semaphore

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
 end:    do
         :: sem!p ->
            sem?v
         od
}

active [5] proctype user()
{
 end:    do
         :: sem?p;
            count++;
            /* critical section */
            count--;
            sem!v
         od
}
```



neither process is intended to terminate
the proper endstate in both proctypes is s_0

the model check can now search for reachable *invalid* end-states

Semaphore example: result

```
$ spin -a semaphore.pml
$ cc -o pan pan.c
$ ./pan

(Spin Version 4.2.6 -- 27 October 2005)
            + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 40 byte, depth reached 5, errors: 0
      16 states, stored
       5 states, matched
      21 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)

unreached in proctype semaphore
        line 13, state 6, "-end-"
        (1 of 6 states)
unreached in proctype user
        line 24, state 8, "-end-"
        (1 of 8 states)
```

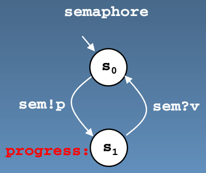
- There are no errors: no invalid end state
- At the end "*unreached ... line 13*" and "*unreached ... line 24*" show that non of the processes terminates (they don't reach the ending "}")

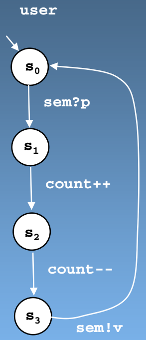
Progress-state labels

- remember Büchi-acceptance
- livelock
- progress-state labels: used to check that the process is really making progress, not just idling or waiting for other processes to make progress
- **every potentially infinite** execution cycle permitted by the model passes through at least one of its progress labels
- If the verifier find cycles **without** the above property: report non-progress loop –corresponding to possible *starvation*
- So, what Spin does is to check for the **absence** of non-progress cycles[10]
- Note: enabling the search for **non-progress** properties (a *liveness* property) automatically disable the search for invalid end states (a *safety* property)
- for simulation runs, such labels have no meaning.

[10] The verifier needs to be compiled with the special option `-DNP`.

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
     do
     :: sem!p ->
progress: sem?v
     od
}

active [5] proctype user()
{
     do
     :: sem?p ->
          count++;
          /* critical section */
          count--;
          sem!v
     od
}
```





we make effective progress each time a user gains access to the critical section: each time state s_1 is reached in proctype semaphore

the model checker can now search for reachable *non-progress* cycles

Verifier output

see also "never claim"

```
$ spin -a sem-prog.pml
$ cc -DNP -o pan pan.c    # enable non-progress checking
$ ./pan -l                # search for non-progress cycles

(Spin Version 4.2.6 -- 27 October 2005)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 44 byte, depth reached 9, errors: 0
      21 states, stored
       5 states, matched
      26 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)

unreached in proctype semaphore
        line 13, state 6, "-end-"
        (1 of 6 states)
unreached in proctype user
        line 24, state 8, "-end-"
        (1 of 8 states)
```

- There are no errors: no assertion violations nor non-progress cycles were found
- This means the model does **not** permit infinite executions that do not contain infinitely many semaphore v operations

What about fairness?

```
byte x = 2;

active proctype A()
{
    do
    :: x = 3 - x
    od
}

active proctype B()
{
    do
    :: x = 3 - x
    od
}
```

x alternates between values 2 and 1 ad infinitum
each process has just 1 state
no progress labels used just yet: every cycle is a non-progress cycle

```
$ spin -a fair.pml
$ gcc -DNP -o pan pan.c # non-progress cycle detection
$ ./pan -l              # invoke np-cycle algorithm
pan: non-progress cycle (at depth 2)
pan: wrote fair.pml.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)
State-vector 24 byte, depth reached 7, errors: 1
       3 states, stored (5 visited)
       4 states, matched
       9 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
```

Q1: what happens if we mark *one* of the do-od loops with a progress label?
Q2: what happens if we mark *both* do-od loops?

Accept states

- 3rd form of meta labels
- Accept-state labels: usually used in `never` claims, but not necessarily
- By marking a state with a label which start with the prefix `accept` the verifier can be asked to find **all** cycles that **do** pass through at least one of those labels

The implicit correctness claim

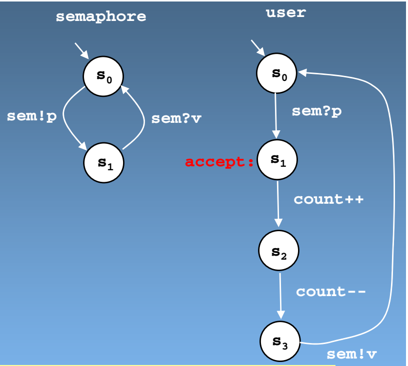
expressed by an accept-state label:

> *There should **not** exist any execution that can pass through an accept-state label infinitely often*

- for simulation: such labels without meaning

Example

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
    do
    :: sem!p ->
          sem?v
    od
}

active [5] proctype user()
{
    do
    :: sem?p ->
accept:   count++;
          /* critical section */
          count--;
          sem!v
    od
}
```



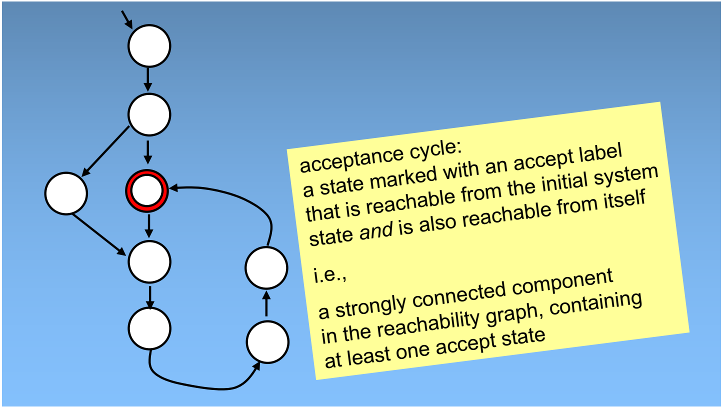
we may want to find infinite executions that do pass through a specially marked state

the state can be marked with an accept label

the model checker can now search for reachable *acceptance* cycles

Acceptance cycles

- Why are they called acceptance cycles?
- It has to do with the automata theoretic foundation we have seen
  - `never` claims (discussed later) formally define ω-automata that accept only those sequences that violate a correctness claim



Fairness assumptions

- default: no assumption about relative speed of executing processes ⇒ counter-examples where a process *pauses indefinitely*
- often: interested in detecting property violations under fairness assumptions
- One of such assumptions is the finite progress assumption: If a process **can** execute a statement, it will eventually proceed with that execution

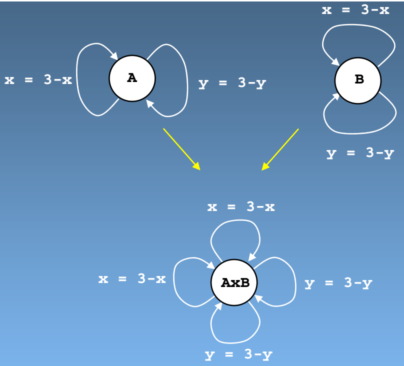
2 degrees

- Weak fairness: If a statement is executable (enabled) infinitely **long**, it will eventually be executed
- Strong fairness: If a statement is executable infinitely **often**, it will eventually be executed

- Several interpretations are still possible – Fairness applied to
  - Non-deterministic statement selection **within** a process
  - Non-deterministic statement selection **between** processes

Statement vs. process selection

```
byte x = 2, y = 2;

active proctype A()
{
    do
    :: x = 3 - x
    :: y = 3 - y
    od
}

active proctype B()
{
    do
    :: x = 3 - x
    :: y = 3 - y
    od
}
```



Spin contains a predefined option for enforcing one specific variant of weak-fairness (run-time option pan -l -f or pan -a -f):

> if a *process* contains at least one statement remains executable infinitely long, that *process* will eventually execute

applies only to *infinite* executions (cycles)

Enforcing fairness constraints

- built-in notion of fairness: **only** to process scheduling
  - not to the resolution of non-deterministic choices inside processes
- But: any type of fairness can be expressed in LTL
- adding fairness assumptions increases the cost of verification
- strong fairness constraints: more costly than weak
  - Weak: *linear* penalty in the number of active processes
  - Strong: *quadratic* penalty in the number of active processes

Never claims

- limitations of previous “claim” mechanisms
- reasoning about *executions*

Example

The truth of p is followed (within a finite number of steps) by the truth of $\neg q$

- two “approaches” that do not work:
  - assertions
  - using an extra process[11] for global invariant checking
- one that would work: LTL (but not here/now)

here

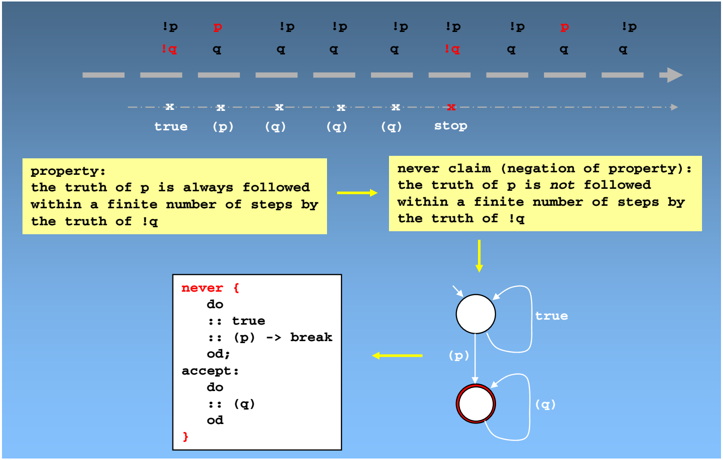
Never claims

manual construction of a “Büchi automaton observer”, using `accept`-labels

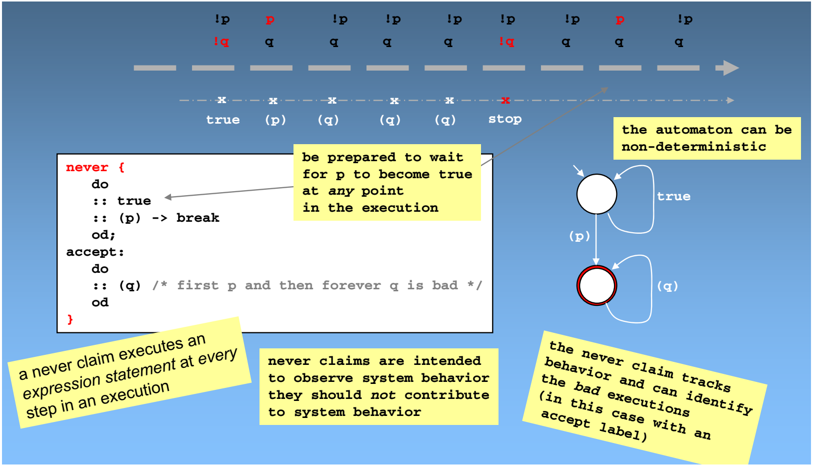
[11] Like `active proctype invariant { ...}`.

Never claims: example

- A never claim defines an observer process executing **synchronously** with the system
- cf. the accept label



- The checker must execute **synchronously** with the system!



Never Claims

- actually: slight misnomer, I think.
- can be non-deterministic
- **all** control flow constructs allowed including
  `if, do, unless, atomic, d_step, goto`
- but: **no side-effect** expression statements[12]
- to define **invalid** execution sequences
- It cannot block
  - A block would mean that the pattern expressed cannot be matched
  - The `never` claim process gives up trying to match the current execution sequence, backs up and tries to match another
  - Pausing in the never claim must be represented explicitly with a self-loop on true
- error found: when
  - closing curly brace of never claim is reached
  - acceptance cycle is closed

[12] `q?[ack]` or `nfull(q)` is okay, but not `q?ack` or *q!ack*

- never claim: slight misnomer
- easiest never claims for: invariant checking: $S \models \Box p$

Observing never claim process

"never I want to observe the opposite of p and if I do I will report it as violation"

```
never {
    do
    :: !p -> break
    :: else
    od
}
```

Never Claims

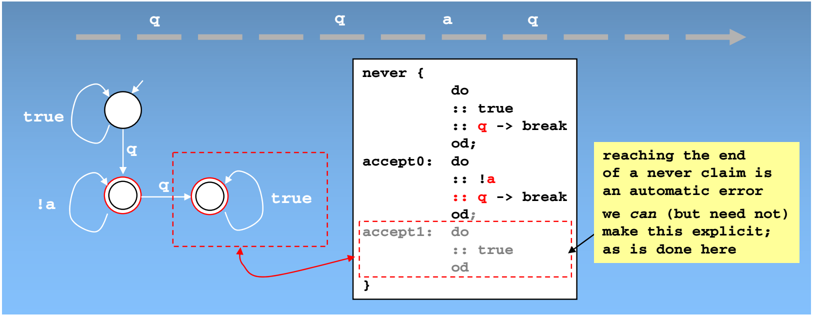
- Convention: use `accept`-state labels **only** in `never` claims and `progress` and `end`-state labels **only** in the behavior model
- Special precautions are needed if non-progress conditions are checked in combination with never claims
  - non-progress is normally encoded in Spin as a predefined never claim

Scope of never claims

- `never` claim: defined **globally**
- Within a claim we can therefore refer to:
  - global variables
  - message channels (using poll statements)
  - process control-flow states (remote reference operations)
  - predefined global variables such as `timeout`, `_nr_pr`, `np_` but **not** process local variables
- In general, we can not refer to events, only to properties of states
  - The effect of an event has to be made visible in the state of the system to become visible to a claim
  - Only *trace assertions* can refer to send/recv events...

Another example: questions and answers

- "Question q is always eventually followed by answer a (assume q and a are properties of states) BEFORE the next question is asked"
- This requirement is violated by any execution where a q is not followed by an a at all, AND by any execution where a q follows a q without an a in between



Example: some conventions

```
never {
            do
            :: true
            :: q -> break
            od;
accept0:    do
            :: !a
            :: q -> break
            od;
accept1:    do
            :: true
            od
}
```

↔

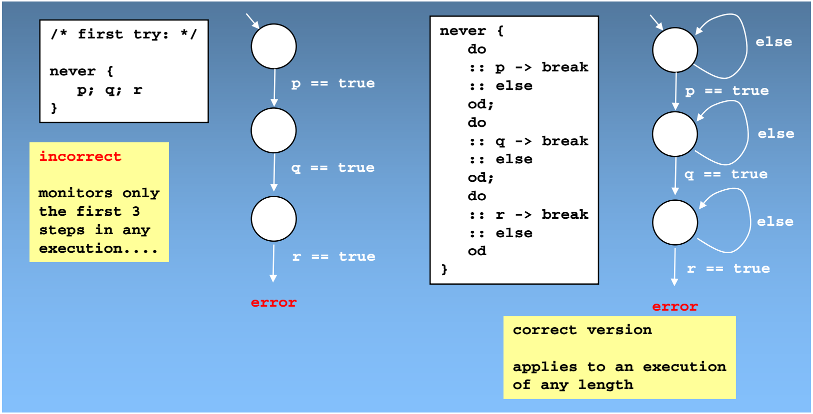
```
never {
            do
            :: true
            :: q -> break
            od;
accept0:    do
            :: !a
            :: q -> break
            od
}
```

reaching the closing curly brace of a never claim means that the entire behavior pattern that was expressed was *matched*, and is always interpreted as an error (it should *never* happen)

never claims are designed to '*accept*' bad behavior – property *violations*

Another example

- "There is no execution where first p becomes *true*, then q, and then r"



Obtaining a Never Claim from an LTL Formula

- `never` claims can be obtained from LTL formula
- The `never` claim automaton of the (negated) formula `![](p -> <>!q)` can be obtained by executing the following Spin command:

  ```
  spin -f '![](p -> <>!q)'
  ```

- Alternatively,
  - You can use the *timeline editor* (see Holzmann's Chap. 13), or
  - You can use the LTL 2 BA fast algorithm from LTL to Büchi Automata `ltl2b -f '![](p -> <>!q)'` (not distributed with Spin, see `http://www.liafa.jussieu.fr/ oddoux/ltl2ba/`)
- `never` claims are equally expressive as ω-word automata (and Büchi automata), so they are more expressive than LTL

Yet another useful form of claims: Trace assertions

- so far: focus on states, more precisely state properties, not events.[13]
- Spin's target application area: protocol/software verification
- concurrent programs with message passing

Particularly important events

channel send and receive

- specific kind of observer process just for those
- keyword `trace`

[13] In the program model. Remember also: in the LTL construction, the Büchi-automaton is labeled by sets of properties. For the Kripke structure/transition system, the states have properties/ are "labelled" by properties. In a way, the system being in a state is a kind of "events" from the perspective of the observing Büchi-automaton.

Trace Assertions

- Trace assertions can be used to reason about valid or invalid sequences of *send* and *receive* statements

```
mtype = { a, b };

chan p = [2] of { mtype };
chan q = [1] of { mtype };

trace {
   do
   :: p!a; q?b
   od
}
```

this assertion only claims something about how send operations on channel *p* relate to receive operations on channel *q*

it claims that every send of a message *a* to *p* is followed by a receive of a message *b* from *q*

a deviation from this pattern triggers an error

if at least one send (receive) operation on a channel q appears in the trace assertion, *all* send (receive) operations on that channel q must be covered by the assertion

cannot use *variables* in trace assertions

cannot use any statement other than send or receive statements in trace assertions

can use q?_ to specify an *unconditional* receive

Notrace Assertions

- A `notrace` assertion states that a particular access pattern is impossible (it reverses the claim) invalid sequences of *send* and *receive* statements

```
mtype = { a, b };

chan p = [2] of { mtype };
chan q = [1] of { mtype };

notrace {
   if
   :: p!a; q?b
   :: q?b; p!a
   fi
}
```

this notrace assertion claims that there is *no execution* where the send of a message a to channel p is followed by the receive of a message b from q, or vice versa: it claims that there must be intervening sends or receives to break these two patterns of access

the notrace assertion is fully matched when the closing curly brace is reached

Correctness claims

Devil's advocate

- All correctness properties that can be verified with Spin can be interpreted as formal claims that certain types of behavior are, or are not, possible
- instead of "verifying" a property, Spin hunts for counter-examples (more efficient as well)

- An assertion formalizes the claim
  - It is impossible for the given expression to evaluate to false when the assertion is reached
- An end-state label formalizes the claim
  - It is impossible for the system to terminate without all active processes having either terminated, or having stopped at a state that was marked with an end-state label
- A progress-state label formalizes the claim
  - It is impossible for the system to execute forever without passing through at least one of the states that was marked with a progress-state label infinitely often

Correctness Claims (cont.)

- An accept-state label formalizes the claim
  - It is impossible for the system to execute forever while passing through at least one of the states that was marked with an accept-state label infinitely often
- A never claim formalizes the claim
  - It is impossible for the system to exhibit the behavior (finite or infinite) that completely matches the behavior that is specified in the claim
- A trace assertion formalizes the claim
  - It is impossible for the system to exhibit behavior that does not completely match the pattern defined in the trace assertion

References I

[Holzmann, 2003] Holzmann, G. J. (2003).
The Spin Model Checker.
Addison-Wesley.

[Morris, 1968] Morris, R. (1968).
Scatter storage techniques.
Communications of the ACM, 11(1):38–44.