Logic Model Checking

Lecture Notes 14:18

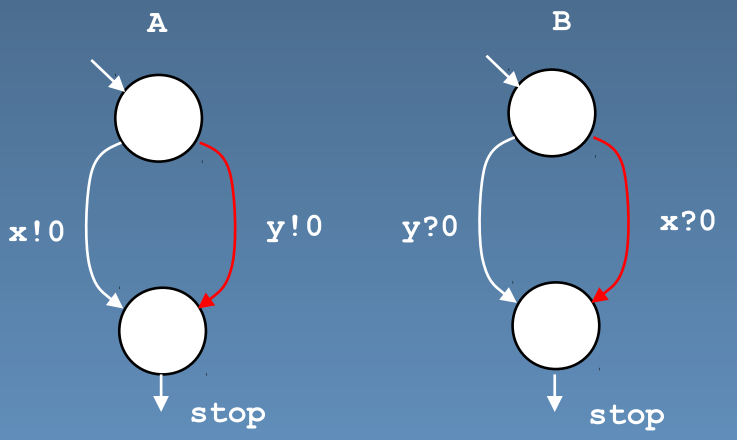
Caltech 118

January-March 2006

Course Text:
The Spin Model Checker: Primer and Reference Manual
Addison-Wesley 2003, ISBN 0-321-22862-6, 608 pgs.

Promela semantics

```
chan x = [0] of { bit };
chan y = [0] of { bit };

active proctype A()
{
	x!0 unless y!0
}

active proctype B()
{
	y?0 unless x?0
}
```



what precisely does this mean?
what are the possible executions?

red arrows take priority over white arrows...

two rendezvous handshakes seem possible:

```
          y!0 <-> y?0
and
          x!0 <-> x?0
```

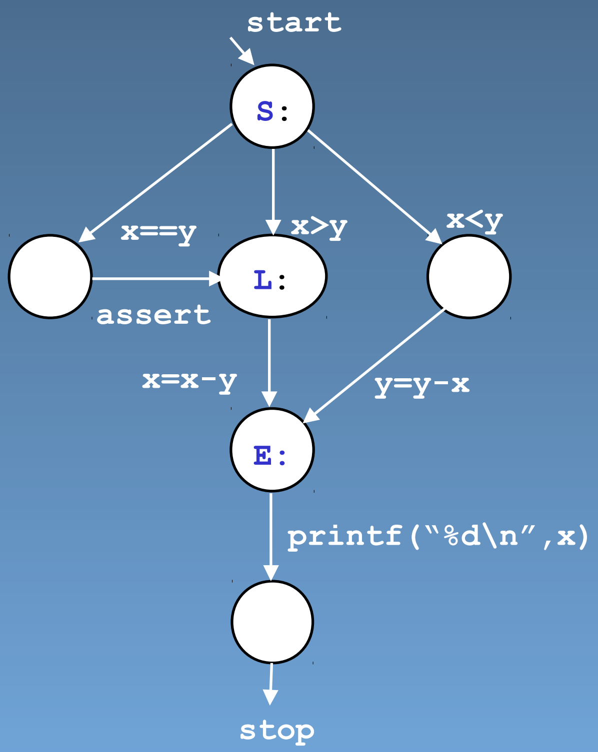
can you tell which can happen without running Spin....?

if you use this method and get a different answer from the Spin *simulator* and the Spin *verifier*, which answer is right?

the semantics of Promela

proctypes and automata

```
active proctype not_euclid()
{
S: if
   :: x == y ->    assert(x != y); goto L
   :: x > y  -> L: x = x - y
   :: x < y  ->    y = y - x
   fi;
E: printf("%d\n", x)
}
```



a Spin model defines a system of:
  states and *state transformers* (transitions)

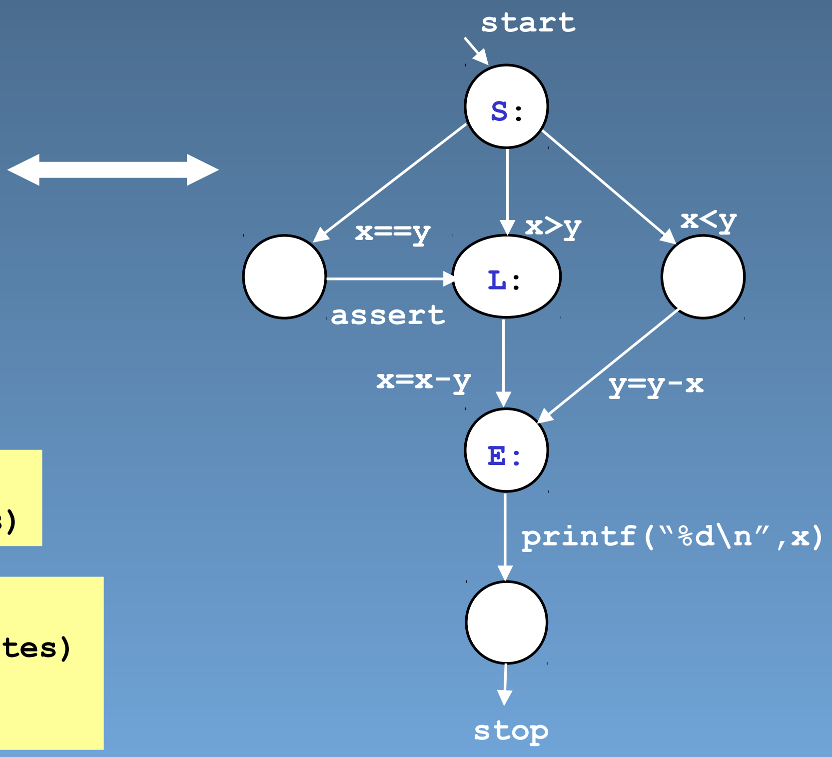
state is maintained in
  sets of process counters (*control* flow states)
  local and global variables and
  message channels

`';'`, `'->'`, `if-fi`, `do-od`, `goto`, etc. are only used to
  define the *transition structure*
  (not the state transformers themselves)
  the only *state transformers* are the *basic* statements:
    `assignment`, `(expr)`, `printf`, `assert`, `send`, `receive`

the semantics of Promela

proctypes and automata

```
active proctype not_euclid()
{
S: if
   :: x == y ->    assert(x != y); goto L
   :: x > y  -> L: x = x - y
   :: x < y  ->    y = y - x
   fi;
E: printf(“%d\n”, x)
}
```



a Spin model defines a system of:
states and *state transformers* (transitions)

state is maintained in
sets of process counters (*control* flow states)
local and global variables and
message channels

‘;’, ‘->’, if-fi, do-od, goto, etc. are only used to
define the *transition structure*
(not the state transformers themselves)
the only *state transformers* are the *basic* statements:
assignment, (expr), printf, assert, send, receive

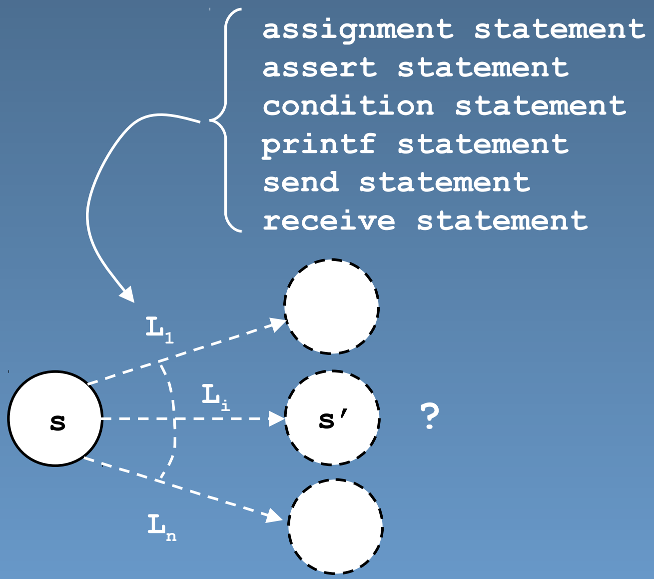
operational model

- to define the semantics of the modeling language, we can define an operational model in terms of *states* and *state transformers (transitions)*
  - we have to define what a "*global system state*" is
  - we have to define what a "*state transition"* is
    - i.e., how the '*next-state*' relation is defined

- *global system states* are defined in terms of a small number of primitive objects:
  - we have to define: variables, messages, message channels, and processes

- *state transitions* are defined with the help of
  - basic statements that label transitions
    - the alphabet of the underlying automata
    - there are only 6 types of labels in the alphabet: assignment, condition, etc.
  - we have to define: transitions, transition selection, and transition execution

transitions



given an arbitrary global state of the system, determine the set of possible immediate successor states

this means determining 3 things:

1. statement executability rules
2. statement selection rules
3. the effect of a statement execution

we only have to define single-step semantics to define the full language

the 3 parts of the semantics definition are defined over 4 types of objects:

1. variables
2. messages
3. message channels
4. processes

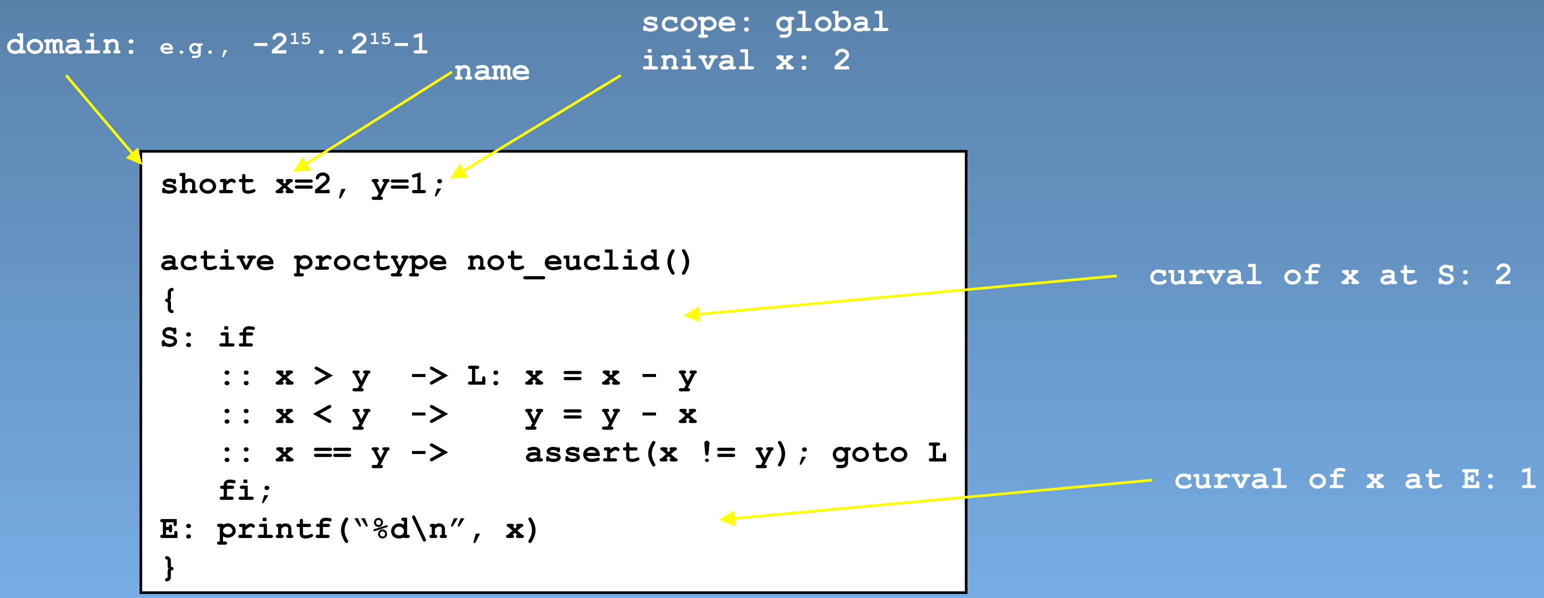
we'll define these first

operational model

variables, messages, channels, processes, transitions, global states

- a promela *variable* is defined by a five-tuple

  {name,scope,domain,inival,curval}



domain: e.g., $-2^{15}..2^{15}-1$

name

scope: global
inival x: 2

```
short x=2, y=1;

active proctype not_euclid()
{
S: if
   :: x > y  -> L: x = x - y
   :: x < y  ->    y = y - x
   :: x == y ->    assert(x != y); goto L
   fi;
E: printf(“%d\n”, x)
}
```

curval of x at S: 2

curval of x at E: 1

operational model

variables, *messages*, channels, processes, transitions, global states

- a *message* is a finite, ordered set of variables

  (messages are stored in channels – *defined next*)

```
mtype = { req, resp, ack };

chan q = [2] of { mtype, bit };

active proctype not_very_useful()
{   bit p;

    do
    :: q?req,p  -> q!resp,p
    :: q?resp,p -> q!ack,1-p
    :: q?ack,_
    :: timeout -> break
    od
}
```

place names for values held in message channel:
slot$_1$.field$_1$
slot$_1$.field$_2$
domains:
mtype
bit

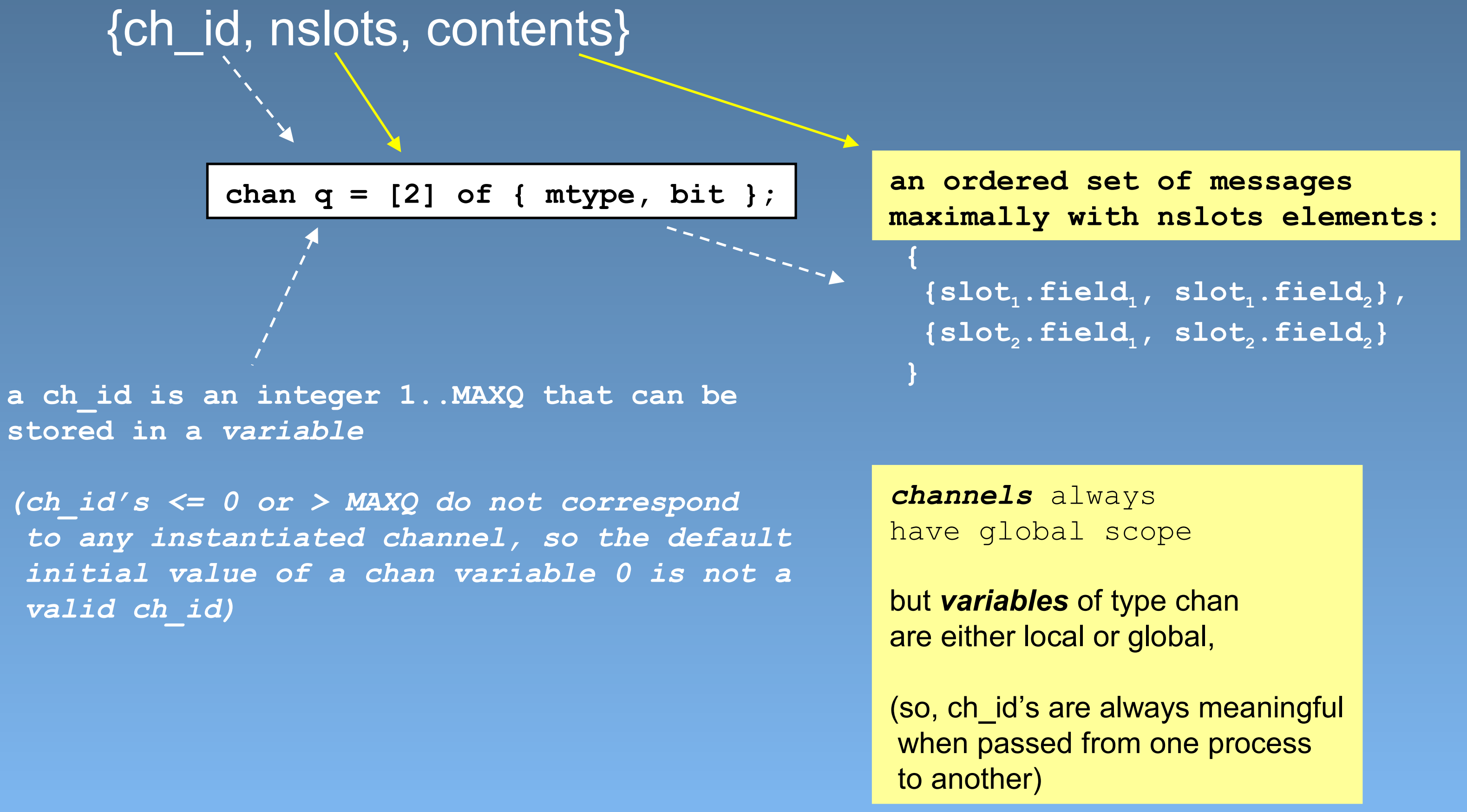
parallel value assignment

operational model

variables, messages, *channels*, processes, transitions, global states

- a *message channel* is defined by a 3-tuple

{ch_id, nslots, contents}



`chan q = [2] of { mtype, bit };`

an ordered set of messages maximally with nslots elements:

{
{slot$_1$.field$_1$, slot$_1$.field$_2$},
{slot$_2$.field$_1$, slot$_2$.field$_2$}
}

a ch_id is an integer 1..MAXQ that can be stored in a *variable*

(ch_id's <= 0 or > MAXQ do not correspond to any instantiated channel, so the default initial value of a chan variable 0 is not a valid ch_id)

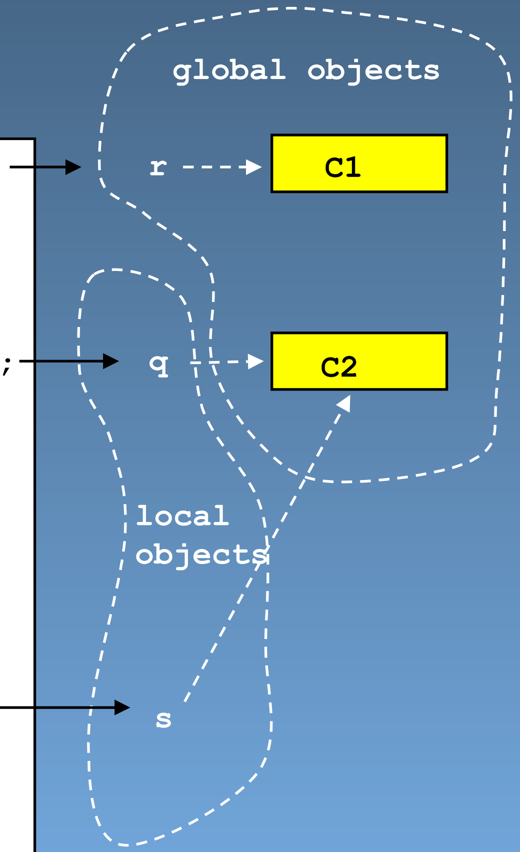
channels always have global scope

but ***variables*** of type chan are either local or global,

(so, ch_id's are always meaningful when passed from one process to another)

channel scope

```
chan r = [0] of { chan };


active proctype A()
{
   chan q = [0] of { int };

   r!q;
   q?100
}


active proctype B()
{
   chan s;

   r?s;
   s!100
}
```



global variable r
holds the ch_id of C1

C1 never disappears

local variable q in A
holds the ch_id of C2
(a global object)

C2 is *created* when A()
is instantiated
it *disappears* when A()
disappears

local variable s in B
is initialized to 0
(not a valid ch_id)

it is set to C2 in the
receive from r == C1

C2 is a globally visible
object with a limited
lifetime...

in the initial system state
r, q, s, C1, and C2 all exist
r points to C1, q points to C2

operational model

variables, messages, channels, *processes*, transitions, global states

- a *process* is defined by a six-tuple

{pid, lvars, lstates, inistate, curstate, transitions}

the current state

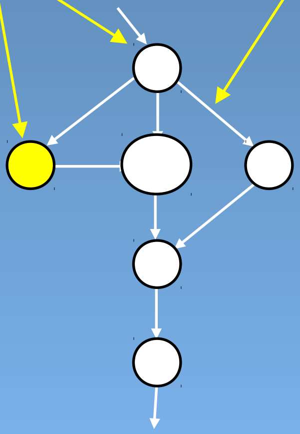
the initial state

a finite set of *transitions* (to be defined) between elements of lstates

process instantiation number

a finite set of integers defining local proc states

finite set of local variables



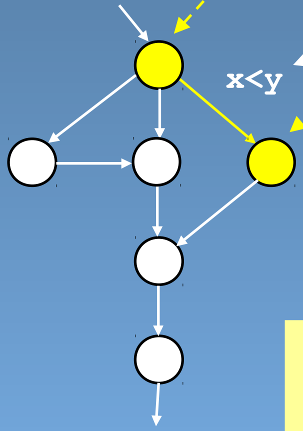
```
process p
          p.curstate
          p.pid
          etc.
```

operational model

variables, messages, channels, processes, *transitions*, global states

- a *transition* is defined by a seven-tuple

  {tri_id, source-state, target-state, cond, effect, priority, rv}



`predefined system variables that are used to define the semantics of unless and rendezvous`

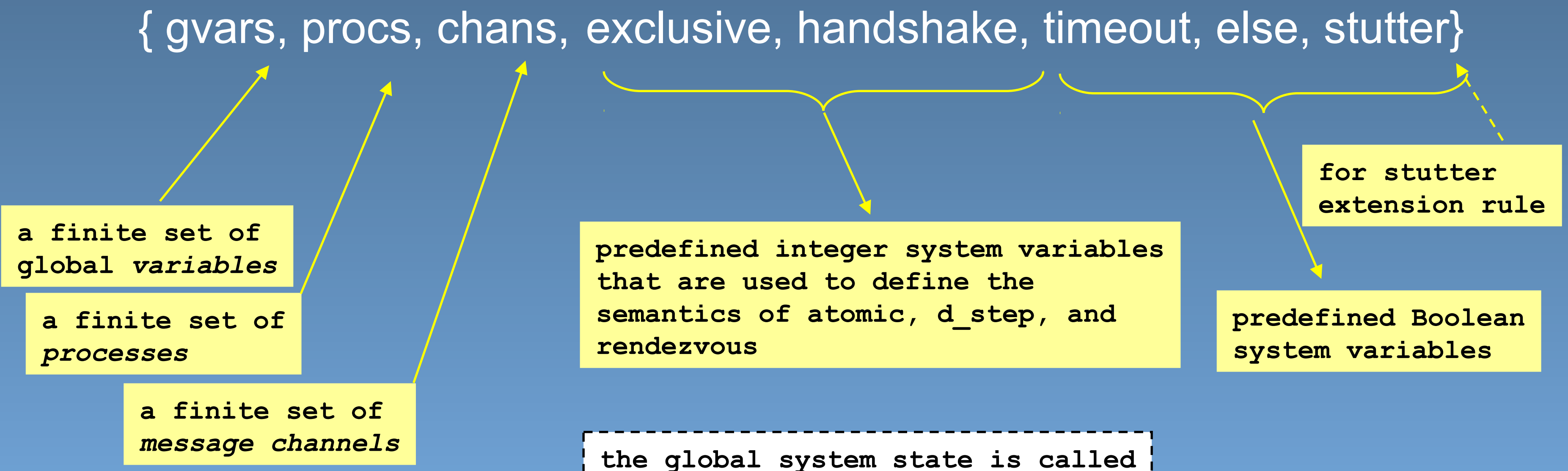
`condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states`

```
transition t:
          t.source
          t.target
          t.cond
          t.effect
          etc.
```

operational model

variables, messages, channels, processes, transitions, *global states*

- a *global state* is defined by a eight-tuple



{ gvars, procs, chans, exclusive, handshake, timeout, else, stutter}

a finite set of global *variables*

a finite set of *processes*

a finite set of *message channels*

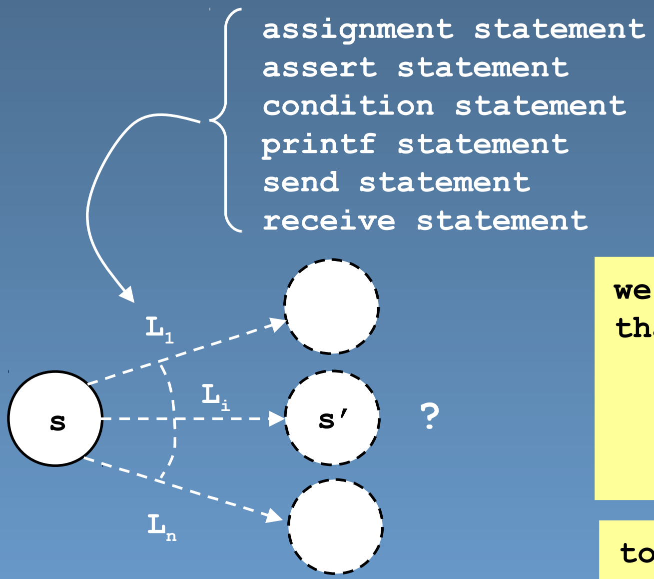
predefined integer system variables that are used to define the semantics of atomic, d_step, and rendezvous

predefined Boolean system variables

for stutter extension rule

the global system state is called the system "*state vector*"

one-step semantics



assignment statement
assert statement
condition statement
printf statement
send statement
receive statement

given an arbitrary global state of the system, determine the set of possible immediate successor states

we′ve defined the only 4 types of objects that hold state:

1. variables
2. messages
3. channels and
4. processes

to define a one-step semantics, we have to define 3 more things:

1. transition executability rules
2. transition selection rules
3. the effect of transition

we do so by defining an algorithm:
an implementation-independent semantics interpretation “engine” for Spin

defining the *next-state* relation

the Promela *semantics engine*

the easy part:
state updating

```
global states s, s'
processes     p, p'
transitions   t, t'
E a set of pairs {p,t}

while ((E = executable(s)) != {})
{
        for-some (process p and transition t) from E
        {     s' = apply(t.effect, s)


                {     s = s'
                      p.curstate = t.target












        }     }
}
while
```

to be defined

executability rules

the hard part:
transition selection

```
global states s, s'
processes     p, p'
transitions   t, t'
Set
executable(State s)
{   new Set E
    new Set e
    [redacted]
AllProcs:

    [redacted]
    [redacted]

    return E      /* executable transitions */
}
```

```
     for each active process p
     {    [redacted]
                {    e = {};  [redacted]
OneProc:             for each transition t in p.trans
                     {    if (t.source == p.curstate
                          [redacted]
                          and   eval(t.cond) == true)
                          {      add (p,t) to set e
                     }    }
                     [redacted]
                          add all elements of e to E
                     [redacted]
        }  [redacted]  }
```

next:
extensions for
timeout, else
rendezvous, atomic,
unless, stutter

executability rules

the hard part:
transition selection

```
global states  s, s'
processes      p, p'
transitions    t, t'
Set
executable(State s)
{   new Set E
    new Set e

    E = {}
    timeout = false
AllProcs:

    [redacted]
    if (E == {} and timeout == false)
    {   timeout = true
        goto AllProcs
    }

    return E      /* executable transitions */
}
```

```
    for each active process p
    {   [redacted]
            {   e = {}; [redacted]
OneProc:        for each transition t in p.trans
                {   if (t.source == p.curstate
                    [redacted]
                        and  eval(t.cond) == true)
                        {     add (p,t) to set e
                }       }
                [redacted]
                        add all elements of e to E
                [redacted]
    }  [redacted]  }
```

next:
extension for **else**

executability rules

the hard part:
transition selection

```
global states s, s'
processes     p, p'
transitions   t, t'
 1 Set
 2 executable(State s)
 3 {   new Set E
 4     new Set e
 5
 6     E = {}
 7     timeout = false
 8 AllProcs:
 9
29
30     [redacted]
31
32
33
34
35     if (E == {} and timeout == false)
36     {    timeout = true
37          goto AllProcs
38     }
39
40     return E      /* executable transitions */
41 }
```

```
 9     for each active process p
10     {   [redacted]
11
12
13           {    e = {};   else = false
14 OneProc:       for each transition t in p.trans
15                {    if (t.source == p.curstate
16                     [redacted]
17
18
19                     and  eval(t.cond) == true)
20                     {      add (p,t) to set e
21                }    }
22                if (e != {})
23                {    add all elements of e to E
24                     continue  /* on to next process */
25                } else if (else == false)
26                {    else = true
27                     goto OneProc
28                }    [redacted]
29     }  [redacted]  }
```

next:
extension for rendezvous semantics

adding semantics for rendezvous 1:2

the predefined variable handshake

```
global states s, s'
processes     p, p'
transitions   t, t'

while ((E = executable(s)) != {})
{
     for-some (process p and transition t) from E
     {    s' = apply(t.effect, s)  ←——————  can set handshake

          if (handshake == 0)
          {    s = s'
               p.curstate = t.target
          } else
          {     /* try to complete a rv handshake */
               E' = executable(s')
               /* if E' is {}, s is unchanged  */

               for-some (process p' and transition t') from E'
               {    s = apply(t'.effect, s')
                    p.curstate  = t.target
                    p'.curstate = t'.target
               }
               handshake = 0
     }     }
}
```

adding semantics for rendezvous 2:2

the predefined variable handshake

```
global states s, s'
processes     p, p'
transitions   t, t'
 1 Set
 2 executable(State s)
 3 {   new Set E
 4     new Set e
 5
 6     E = {}
 7     timeout = false
 8 AllProcs:
 9
29
30  [redacted]
31
32
33
34
35     if (E == {} and timeout == false)
36     {    timeout = true
37          goto AllProcs
38     }
39
40     return E      /* executable transitions */
41 }
```

```
 9     for each active process p
10     {  [redacted]
11
12
13            {    e = {};  else = false
14 OneProc:        for each transition t in p.trans
15                 {    if (t.source == p.curstate
16                      [redacted]
17                      and (handshake == 0
18                      or   handshake == t.rv)
19                      and  eval(t.cond) == true)
20                      {      add (p,t) to set e
21                 }    }
22                 if (e != {})
23                 {    add all elements of e to E
24                      break    /* on to next process */
25                 } else if (else == false)
26                 {    else = true
27                      goto OneProc
28                 }  [redacted]
29        }  [redacted]  }
```

next:
extensions for atomic and d_step sequences

adding semantics for atomic sequences

the predefined variable exclusive

```
global states s, s'
processes     p, p'
transitions   t, t'

 1 Set
 2 executable(State s)
 3 {   new Set E
 4     new Set e
 5
 6     E = {}
 7     timeout = false
 8 AllProcs:
 9
29
30
31     if (E == {} and exclusive != 0)
32     {   exclusive = 0
33         goto AllProcs
34     }
35     if (E == {} and timeout == false)
36     {   timeout = true
37         goto AllProcs
38     }
39
40     return E      /* executable transitions */
41 }
```

```
 9     for each active process p
10     {  if (exclusive == 0
11        or  exclusive == p.pid)
12        {
13            {    e = {};   else = false
14 OneProc:        for each transition t in p.trans
15                 {    if (t.source == p.curstate
16
17                      and (handshake == 0
18                      or   handshake == t.rv)
19                      and  eval(t.cond) == true)
20                      {     add (p,t) to set e
21                 }    }
22                 if (e != {})
23                 {    add all elements of e to E
24                      break    /* on to next process */
25                 } else if (else == false)
26                 {    else = true
27                      goto OneProc
28                 }
29      }   }   }
```

next:
extension for unless sequences

adding semantics for unless

statement priority levels

```
global states s, s'
processes     p, p'
transitions   t, t'
 1 Set
 2 executable(State s)
 3 {   new Set E
 4     new Set e
 5
 6     E = {}
 7     timeout = false
 8 AllProcs:
 9
29
30
31     if (E == {} and exclusive != 0)
32     {    exclusive = 0
33          goto AllProcs
34     }
35     if (E == {} and timeout == false)
36     {    timeout = true
37          goto AllProcs
38     }
39
40     return E       /* executable transitions */
41 }
```

```
 9     for each active process p
10     {   if (exclusive == 0
11         or  exclusive == p.pid)
12         {  for u from high to low    /* priority */
13               {    e = {};  else = false
14 OneProc:           for each transition t in p.trans
15                    {    if (t.source == p.curstate
16                         and t.prty == u
17                         and (handshake == 0
18                         or   handshake == t.rv)
19                         and  eval(t.cond) == true)
20                         {     add (p,t) to set e
21                    }    }
22                    if (e != {})
23                    {    add all elements of e to E
24                         break    /* on to next process */
25                    } else if (else == false)
26                    {    else = true
27                         goto OneProc
28                    }  /* else lower the priority */
29      }   }   }
```

next:
adding the stutter extension rule

the stutter extension rule

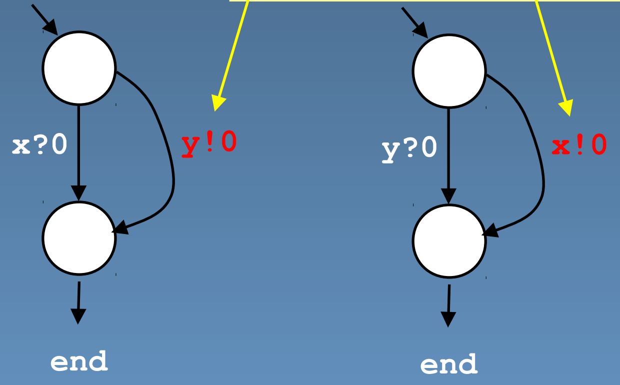
```
global states s, s'
processes     p, p'
transitions   t, t'

while ((E = executable(s)) != {})
{
     for some (process p and transition t) from E
     {    s' = apply(t.effect, s)

          if (handshake == 0)
          {    s = s'
               p.curstate = t.target
          } else
          {     /* try to complete rv handshake */
               E' = executable(s')
               /* if E' is {}, s is unchanged  */

               for some (process p' and transition t') from E'
               {    s = apply(t'.effect, s')
                    p.curstate  = t.target
                    p'.curstate = t'.target
                    handshake = 0
                    break
     }     }   }
}
while (stutter) { s = s }  /* stutter extension rule */
```

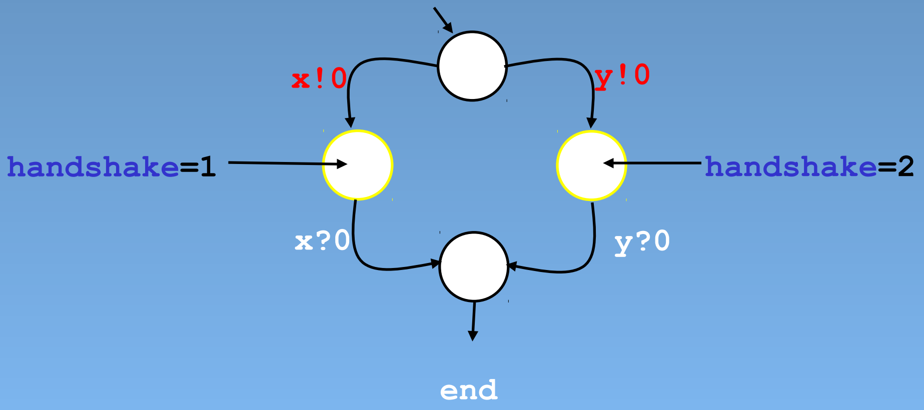
example 1:3

using the semantics engine

```
chan x = [0] of { bit };
chan y = [0] of { bit };

active proctype A()
{
	x?0 unless y!0
}

active proctype B()
{
	y?0 unless x!0
}
```

unless escapes have higher priority





Q: what is the combined system behavior?

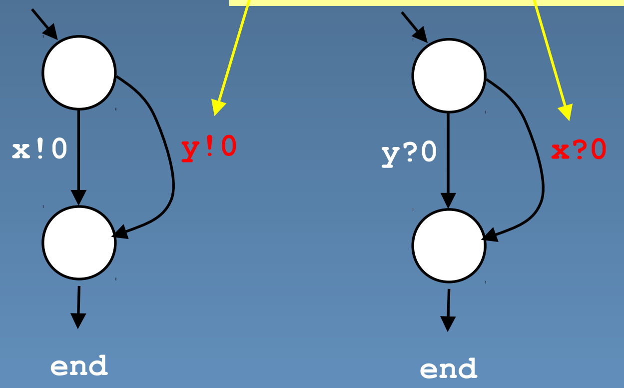
A: a non-deterministic selection between
x!0;x?0
and
y!0;y?0

example 2:3

using the semantics engine

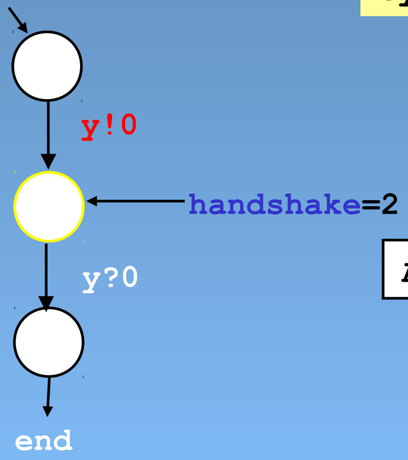
```
chan x = [0] of { bit };
chan y = [0] of { bit };

active proctype A()
{
     x!0 unless y!0
}

active proctype B()
{
     y?0 unless x?0
}
```

unless escapes have higher priority



Q: what is the combined system behavior?

is it
x!0;x?0
or
y!0;y?0
?



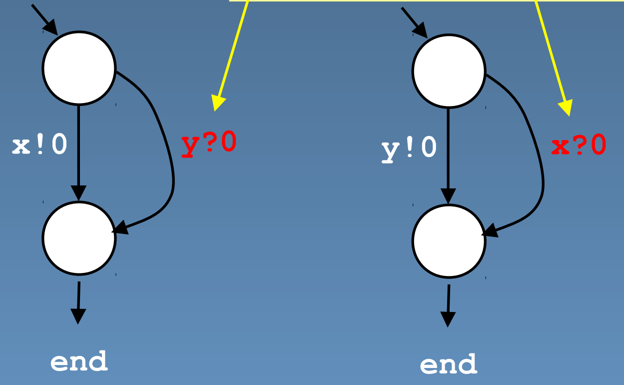
A: only y!0;y?0 can happen

example 3:3

using the semantics engine

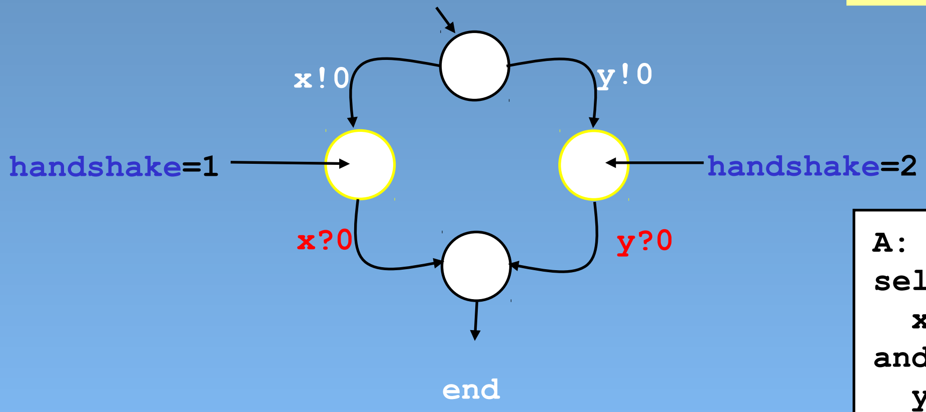
```
chan x = [0] of { bit };
chan y = [0] of { bit };

active proctype A()
{
    x!0 unless y?0
}

active proctype B()
{
    y!0 unless x?0
}
```

unless escapes have higher priority



Q: what is the combined system behavior?

is it
  x!0;x?0
    or
  y!0;y?0
?

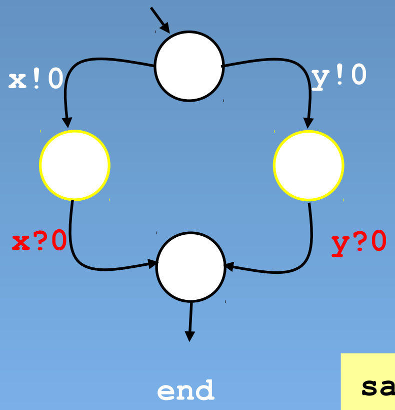


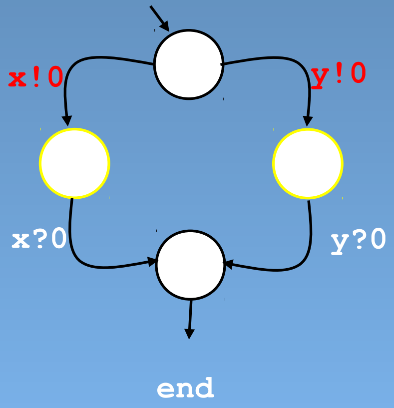
A: a non-deterministic selection between
  x!0;x?0
and
  y!0;y?0

compare

```
chan x = [0] of { bit };
chan y = [0] of { bit };

active proctype A()
{
    x!0 unless y?0
}

active proctype B()
{
    y!0 unless x?0
}
```

```
chan x = [0] of { bit };
chan y = [0] of { bit };

active proctype A()
{
    x?0 unless y!0
}

active proctype B()
{
    y?0 unless x!0
}
```





same global behavior but for very different reasons....

what about never claims, etc.?

meta-semantics

- *correctness properties* do not define new behavior, they just monitor it
  - and complain bitterly when interesting things are seen
- a *verification engine* can make pronouncements on properties of behavior
  - this is at a *higher* level of semantics: it interprets the goodness or badness of a behavior instead of defining the behavior itself
- a *never claim* is designed to *select* those behaviors that could possibly lead to “interesting” behavior
  - the distinction between “good” and “bad”, “interesting” and “uninteresting” is a meta-statement *about* behavior: not part of the behavior itself, and therefore not part of the operational semantics....