The SPIN Model Checker

Metodi di Verifica del Software

Andrea Corradini – GianLuigi Ferrari

Lezione 2

2011

Slides per gentile concessione di Gerard J. Holzmann

process synchronization with provided clauses

```
bool toggle = true;          /* global variable          */
short cnt;                   /* default initial value 0 */

active proctype A() provided (toggle == true )
{
L:      cnt++;               /* increment cnt by 1 */
        printf("A: cnt=%d\n", cnt);
        toggle = false;      /* yield control to B */
        goto L
}

active proctype B() provided (toggle == false)
{
L:      cnt--;               /* decrement cnt by 1 */
        printf("B: cnt=%d\n", cnt);
        toggle = true;       /* yield control to A */
        goto L
}
```

assignment

print statement

assignment

control-flow

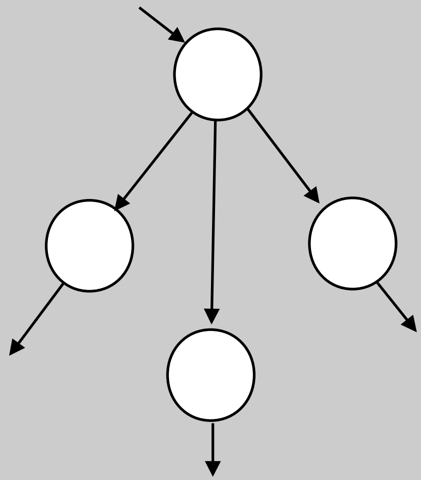
```
$ spin toggle.pml | more
A: cnt = 1
    B: cnt = 0
A: cnt = 1
    B: cnt = 0
A: cnt = 1
    B: cnt = 0
...
```

true == 1
false == 0

- a process can only execute statements if its provided clause evaluates to true
- the default provided clause is true

basic statements

- basic statements define the primitive state transformers in Promela
- they end up labeling the edges (transitions) in the underlying finite state automata
- there is only a very small number of *basic* statements in Promela



states and state transformers

6 types of basic statements

- assignment: x++, x--, x = x+1, x = run P()
- expression statement: (x), (1), run P(), skip, true, else, timeout
- print: printf(“x = %d\n”, x)
- assertion: assert(1+1==2); assert(false)
- send: q!m
- receive: q?m

executability of basic statements

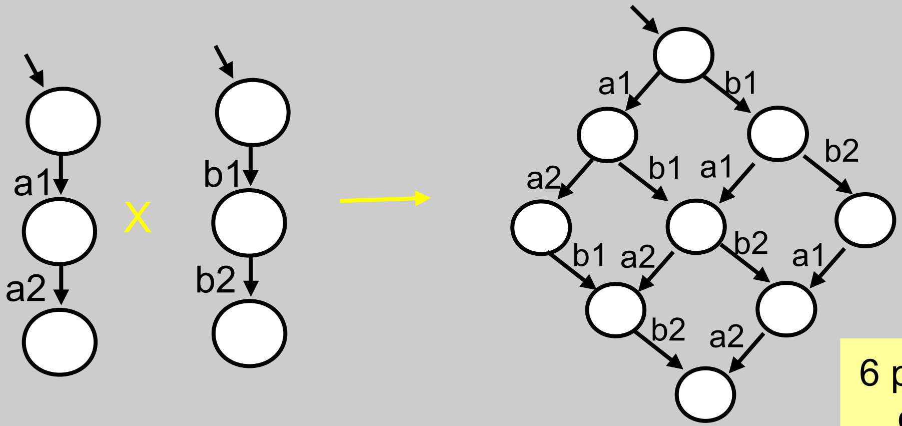
the executability of a statement may depend on the global state of the system

- a Promela statement is either
  - executable   the statement *can* be executed, or
  - blocked   the statement *cannot* be executed (yet)
- 3 types of basic statements we have already seen
  - print statements
    - always unconditionally executable, no effect on state
  - assignment statements
    - always unconditionally executable, changes value of precisely one variable, specified on the left-hand side of the ‘=‘ operator
  - expression statements
    - executable only if expression evaluates to non-zero (*true*)

`2 < 3`	is always executable
`x < 27`	executable iff the value of `x` is less than `27`
`3 + x`	executable iff `x` is not equal to `–3`

statement interleaving

- processes execute concurrently and asynchronously
  - there can be an arbitrarily long pause in between any two statement executions within a process
- process *scheduling* decisions are non-deterministic
- statement executions from different processes are arbitrarily interleaved in time
  - basic statements execute atomically
- local choice within processes can also be non-deterministic



2 levels of nondeterminism:
system level (process selection)
process level (statement selection)

6 possible interleavings
of a1;a2 and b1;b2

executability

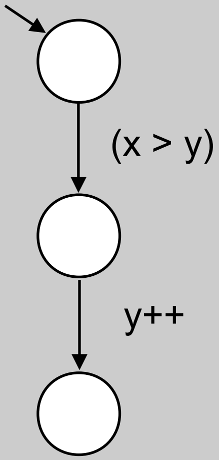
expression statements are first-class citizens in Promela
an expression statement can be used as a synchronizer:
it is executable only if it evaluates to non-zero (true)

where in C one would write:

```
while (x <= y)
    /* wait */;
y++;
```

in Promela this becomes:

```
(x > y) -> y++
```



synchronizer
through executability rule

pseudo statements

- some pseudo-statements:
  - skip – always executable, no effect, same as expression (1)
  - true – always executable, no effect on state, same as expression (1)
- there is no "run *statement*" – run is an *operator* that can appear in restricted expression statements...
  - returns 0 if the max nr of processes would be exceeded by the creation of a new process (the number of processes is bounded)
  - returns the pid of the new process otherwise

```
int x;            /* the default initial value of x is 0 */
proctype A()
{ int y=1;

   skip;
   run B();
   x=2;
  (x>2 && y==1);
   printf("x %d, y %d\n", x, y)
}
```

executable only if **B** can be created

can become executable only if another process changes the value of global variable **x**

run expressions are special

- a run operator can only be used in *special* expressions
- all run-free expressions in Promela are side-effect free
  - they can be evaluated without causing a change of state
  - unlike in C, e.g. where one could say: (x++ <= --y)
- there can be only one run operator in an expression and if there is one, there can be no other clauses; ruling out:
  - (run B() && run A()) could fail with partial side-effect
  - !(run B()) same as expr: (_nr_pr >= 255)
  - run B() && (a > b) could start an arbitrary number of copies of B() while (a <= b)
- it is typically a modeling *error* if run can ever return 0

another type of basic statement (#4)

- assert(*expression*)
  - an *assertion statement* is always executable and has no effect on the state of the system when executed
  - Spin reports a *error* if the expression can evaluate to zero (false),
  - the assertion statement can be used to check *safety properties* (properties of local process states or global system states)

```
int n;

active proctype invariant()
{
    assert(n <= 3)
}
```

this process has only one executable statement – because it is an *asynchronous* process, this statement might be executed at any time – it need not execute immediately
this is precisely the capability we want in verification, when checking a system invariant condition: it should hold no matter when the assertion is checked
the model checker will make sure this is true

example: mutual exclusion

allow only 1 process in a critical section at a time
without relying on a hardware test&set instruction

```
bool busy;                       /* signal entering/leaving the section */
byte mutex;                      /* counts # procs in critical section  */

proctype P(bit i)
{ (!busy)-> busy = true;         /* wait for busy to be false, then set it to true */
  mutex++;
  printf("P%d in critical section\n", i);
  mutex--;
  busy = false;
}
active proctype invariant()
{ assert(mutex <= 1);
}
init {
  atomic { run P(0); run P(1) }
}
```

a potential race condition: both processes can evaluate (**!busy**) before setting it to *false*

no loop required

start two instances of P atomically

a model checking run

```
$ spin -a mutex1
$ gcc -DSAFETY -o pan pan.c
$ ./pan
```

guided simulation of the counter-example that was generated

```
$ spin -t -p mutex1
```

Peterson's algorithm (1981)

```
mtype = { A_Turn, B_Turn };
bool x, y;    /* signal entering/leaving the section  */
byte mutex;   /* # of procs in the critical section   */
mtype turn = A_Turn;   /* who's turn is it?            */

active proctype A()
{ x = true;
  turn = B_Turn;
  (!y || turn == A_Turn) ->
  mutex++;
  /* critical section */
  mutex--;
  x = false;
}

active proctype B()
{ y = true;
  turn = A_Turn;
  (!x || turn == B_Turn) ->
  mutex++;
  /* critical section */
  mutex--;
  y = false;
}

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

basic data types

(book, Table 3.1 p. 41)

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;

3 bits of storage
range 0..7

the default initial value of *all* data objects (global *and local*) is *zero*

all variables (local and global) must be declared before they are used

a variable declaration can appear anywhere…

note: there are no reals, floats, or pointers
deliberately: verification models are meant to
model *coordination* not *computation*

mtype declarations
(originally used for: *m*essage *type* declarations)

- a way to introduce symbolic constant values
- mtype declaration:

```
mtype = { apple, pear, banana, cherry };
mtype = { ack, msg, err, interrupt };  /* up to 255 names total */
```

- declaring variables of type mtype:

```
mtype a;                              /* uninitialized, value 0 */
mtype b = pear;                       /* value always non-zero  */
```

expression evaluation

- all expressions are evaluated in the widest type (int)
- in assignments and message passing operations, the resulting value is mapped (truncated) to the target type *after* evaluation
  - the Spin *simulator* warns if there is loss of information
  - the Spin *parser* rejects only grievous type errors

```
mtype = { apple, pear };

active proctype tryme()
{    byte x;
     short y = 1024;
     chan a, b;
     mtype p;

         a = a+b;     /* no good -- error */
         x = 257;     /* information loss -- warning */
         x = y;       /* information loss -- warning */
         p = y/8;     /* dubious, but no warning...  */
}
```

arrays and user-defined data types

one-dimensional arrays:

```
byte a[27];
bit  flags[4] = 1;
```

all array elements are initialized to the same value (default 0)

as in C, array indices start at 0

user-defined data types:

```
typedef record {
  short f1;
  byte  f2 = 4;
}

record rr;
rr.f1 = 5
```

keyword

name of user-defined data type

default initial value is again 0

declaration of a variable of the newly defined type

reference to a structure element

an indirect way to define multi-dimensional arrays with typedefs and macros

```
typedef array {  byte b[4]; }
array a[4];

a[3].b[2] = 1;
```

or alternatively:

```
#define ab(x,y)     a[x].b[y]

ab(3,2) = ab(2,3) + ab(3,2)
```

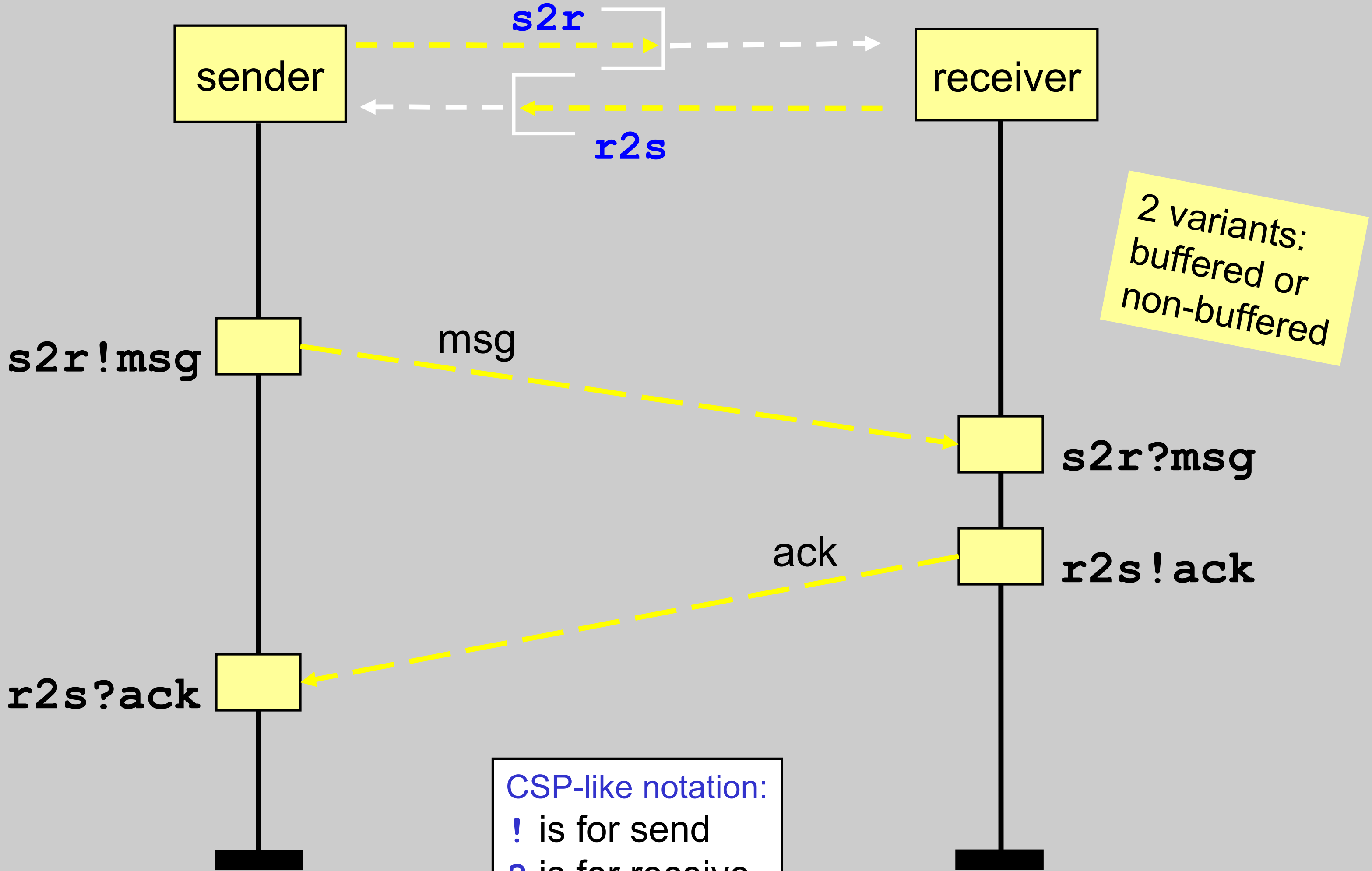
the standard C preprocessor is used to preprocess all models before parsing

supports:

```
    #define ..  ..
    #if ..
    #ifdef ..
    #ifndef ..
    #include "..."
```

etc.

the last two types of basic statements: send and receive



CSP-like notation:
! is for send
? is for receive

message channels

- message passing takes place via *channels* (bounded queues/buffers) either buffered (asynchronously) or unbuffered (by synchronous *rendezvous* handshake)
- sample channel declaration:

```
chan x = [10] of {int, short, bit};
```

type name

variable name

initializer

maximum nr of msgs the channel can store
zero defines a rendezvous channel

structure of messages that can be sent through the channel
a list of type names: one for each field in the message

uninstantiated channel variable a

a rendezvous channel c

```
chan a;
chan c       = [0] of {bit};
chan toR     = [2] of {mtype, bit, chan};
chan line[2] = [1] of {mtype, record};
```

channels can be sent across channels

an array of 2 channels

a user-defined type

send and receive

send: ch!$expr_1$, … $expr_n$

- values of $expr_i$ correspond to the types from the chan declaration
- *executable* if the target channel is *not full*

receive: ch?$const_1$ or var_1, … $const_n$ or var_n

- var_i fields are set to the value from the corresponding field in the message
- $const_i$ fields are constraints on the corresponding fields that must be matched
- *executable* when the target channel is *not empty* and the first message matches all constant fields in the receive

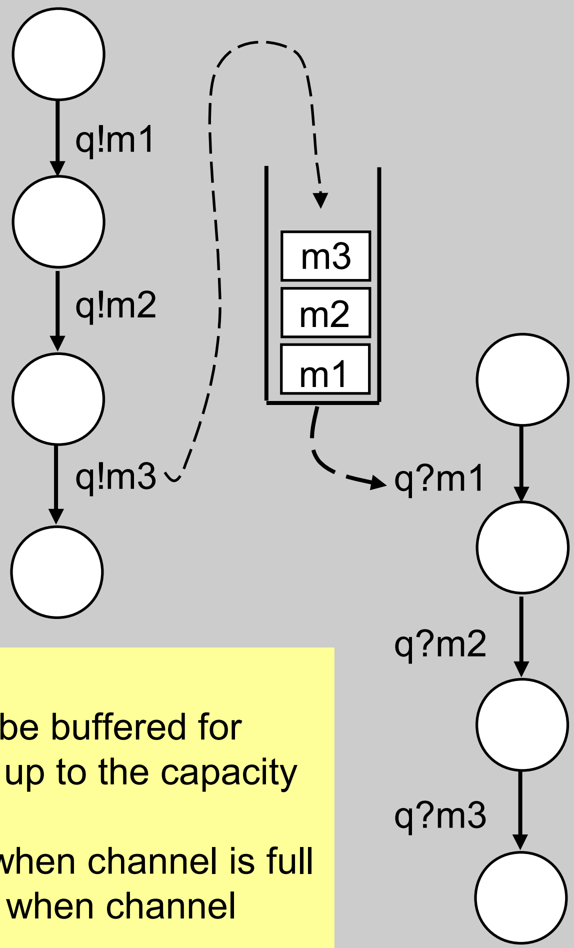
example:

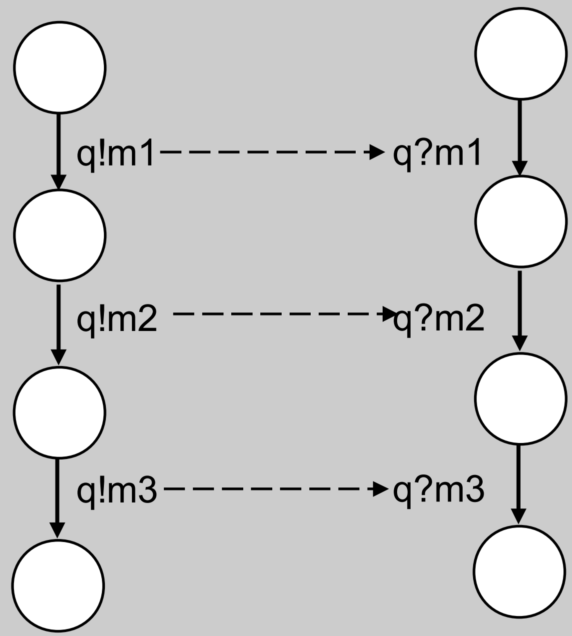
```
#define ack 5
chan ch = [N] of { int, bit };
bit seqno;
ch!ack,0;
ch?ack,seqno
```

alternatively:

```
ch!ack(0);
ch?ack(seqno)
```

asynchronous and synchronous message passing





asynchronous
messages can be buffered for later retrieval – up to the capacity of the channel
sender blocks when channel is full
receiver blocks when channel is empty

synchronous
with channel capacity 0, as in:

chan ch = [0] of { mtype };

can only perform an rv handshake
not store messages
sender blocks until matching receiver is available and vice versa

rendezvous channels

- rendezvous message passing
  - the size of the channel is declared to be zero
  - a send operation is enabled (a send offer) iff there is a matching receive operation that can be executed simultaneously, with all constant fields matching
  - on a match, both send and receive are executed *atomically*
- *example:*

```
chan ch = [0] of {bit, byte};
```

  - P offers: `ch!1,3+7`
  - Q accepts: `ch?1,x`
  - after the rendezvous handshake completes, **`x`** has value **`10`**

message must match value 1
in the first message field,
but can accept any value in the
second message field (x)

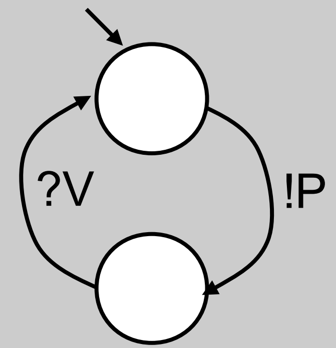
example: modeling a semaphore

```
mtype = { P, V };

chan sema = [0] of { mtype };

active proctype semaphore()
{
L:  sema!P -> sema?V; goto L
}

active [5] proctype user()
{
L:  /* non-critical */
    sema?P ->
    /* critical */
    sema!V;
    goto L
}
```



P – passeren (Dutch)
V - vrijgeven

other operations on channels

- len(q) returns the number of messages in q
- empty(q) true when q is currently empty
- full(q) true when q is filled to capacity
- nempty(q) added to support optimization
- nfull(q) added to support optimization

used instead of !empty(q) or !full(q)
the parser makes this easy to remember:
it rejects the negated forms

brackets, braces
channel poll

- q?[n,m,p]
  - is now a side-effect free Boolean *expression*
  - evaluates to *true* precisely when q?n,m,p is executable, but has *no* effect on n,m,p and does *not* change contents of q

- q?<n,m,p>
  - 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 contents of q

- q?n(m,p)
  - alternative notation for standard receive; same as q?n,m,p
  - sometimes useful for separating type from args

the scope of a chan declaration

- the name of a channel can be local or global, but the channel itself is always a global object....
- this makes obscure things like this work:

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

active proctype A()
{       chan a;             /* uninitialized local channel */

        x?a;                /* get channel id, provided by process B */
        a!x                 /* and start using b's channel! */
}

active proctype B()
{       chan b = [2] of { chan };  /* initialized local channel */

        x!b;                /* make channel b available to A    */
        b?x;                /* value of x doesn't really change */
        0                   /* avoid death of B, or else b dissappears */
}
```

macros – the cpp preprocessor

- all Spin models are by default processed by the standard C preprocessor for *file-inclusion* and *macro expansion*
- typical uses
  - constants

```
#define MAXQ            2
chan q = [MAXQ] of { mtype, chan };
```

```
or:
spin -DMAXQ=2 model
```

  - macros

```
#define RESET(a) \
    atomic { a[0]=0; a[1]=0; a[2]=0; a[3]=0 }
```

  - conditional code

```
#define LOSSY 1
...
#ifdef LOSSY
   active proctype Daemon() { /* steal messages */ }
#endif
...
#if 0
    comments
#endif
```

the scope of a data object

- there are only *two* levels of scope:
  - global (data visible to all active processes)
  - local (data visible to only the process that contains the declaration)
    - there is no sub-scope (e.g., for blocks or *inlines*)
    - the scope of a local variable is *always* the complete proctype body

```
active proctype main()
{       int x, y;     /* x and y declared in outer block */
        {       /* a block: a statement sequence */
                int y, z;               /* error, redeclaration of y */
                x++; y++; z++           /* original y is used */
        };                      /* note semi-colon placements */

        /* variable z remains in scope! */
        printf("y = %d, z = %d\n", y, z)        /* prints: 1, 1 */
}
```

defining control flow

- 5 ways to define control flow structures in proctypes:
  - the obvious: semi-colons, gotos and labels
  - structuring aids:
    - **inlines**
    - **macros**
  - atomic sequences, making things indivisible:
    - atomic { ... }
    - d_step { ... }
  - non-deterministic selection and iteration
    - if .. fi
    - do .. od
  - escape sequences, for error handling/interrupts:
    - { ... } unless { ... }

non-deterministic selection

```
if
:: guard1 -> stmnt1.1; stmnt1.2; stmnt1.3; …
:: guard2 -> stmnt2.1; stmnt2.2; stmnt2.3; …
:: …
:: guardn -> stmntn.1; stmntn.2; stmntn.3; …
fi
```

- if at least one guard is executable, the if statement is *executable*
- if more than one guard is executable, one is selected non-deterministically
- if none of the guard statements is executable, the if statement *blocks*
- *any* type of basic or compound statement can be used as a guard

inspired by Dijkstra's guarded command language,
but the semantics differ: the if does not abort when all guards are unexecutable:
it blocks execution instead

Recommded reading:
E.W. Dijkstra,
Guarded commands, nondeterminacy, and formal derivation of programs.
Comm. ACM, Aug. 1975, Vol. 18, No. 8, pp. 453-457.

the if-statement

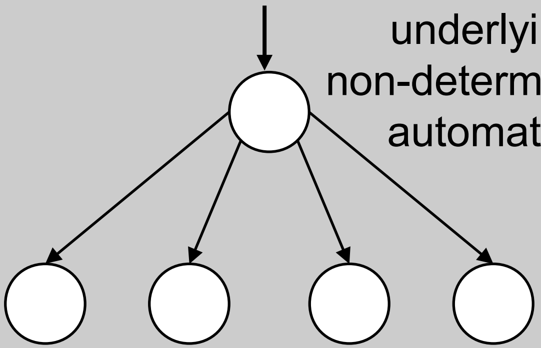
```
/* find the max of x and y */
if
:: x >= y -> m = x
:: x <= y -> m = y
fi
```

```
/* 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.



underlying non-deterministic automaton

the predefined *expression* ‘else’

where in C one writes:

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

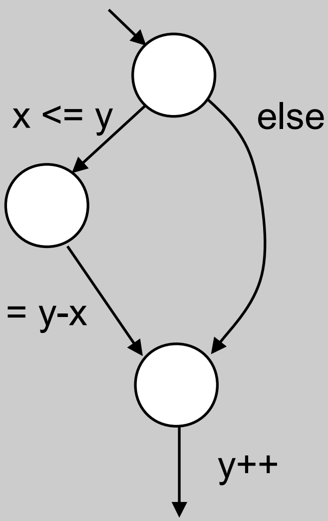
i.e., omitting the ‘else’

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)

timeout

```
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)

Q: could you use 'else' instead of 'timeout' in this context?

timeout and else

- timeout and else are strangely related
  - both are predefined Boolean expressions
  - they evaluate to *true* or *false*, depending on context

- **`else`** is *true* iff
  no other statement in the same *process* is executable
- **`timeout`** is *true* iff
  no other statement in the same *system* is executable

- a timeout can be seen as a system level else
  - *else* cannot be combined with other conditionals
  - *timeout* can be combined, e.g. as in (timeout && a > b)