# Spin

Spin is a model checker which implements the LTL model-checking procedure described previously (and much more besides).

Developed by Gerard Holzmann of Bell Labs

Has won the prestigious ACM Software System Award in 2001
(other winners include Unix, TeX, Java, TCP/IP, Apache, PostScript, ...)

WWW homepage: http://spinroot.com

Literature: Holzmann, The Spin Model Checker
(available in the library of the computer science building,
*Semesterapparat* Esparza)

# Using Spin

Transition systems described with Promela (Protocol Meta Language)

Can describe finite-state systems.

Useful for describing concurrent processes with synchronous and asynchronous communication, variables, advanced datatypes (e.g., records).

# Statements (1)

- The body of a process consists of a sequence of statements. A statement is either
  - executable: the statement can be executed immediately.
  - blocked: the statement cannot be executed.

executable/blocked depends on the global state of the system.

- An assignment is always executable.
- An expression is also a statement; it is executable if it evaluates to non-zero.

| | |
|---|---|
| `2 < 3` | always executable |
| `x < 27` | only executable if value of `x` is smaller `27` |
| `3 + x` | executable if `x` is not equal to `–3` |

# Statements (2)

Statements are separated by a semi-colon: ";".

- The **skip** statement is always executable.
  - "does nothing", only changes process' process counter
- A **run** statement is only executable if a new process can be created (remember: the number of processes is bounded).
- A **printf** statement is always executable (but is not evaluated during verification, of course).

```
int x;
proctype Aap()
{
  int y=1;
  skip;
  run Noot();
  x=2;
  x>2 && y==1;
  skip;
}
```

Executable if Noot can be created…

Can only become executable if a some other process makes **x** greater than 2.

# if-statement (1)

inspired by:
Dijkstra's guarded command language

```
if
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
fi;
```

- If there is at least one **`choice`**$_i$ (guard) executable, the **`if`**-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no **`choice`**$_i$ is executable, the **`if`**-statement is blocked.
- The operator "**`->`**" is equivalent to "**`;`**". By convention, it is used within **`if`**-statements to separate the guards from the statements that follow the guards.

# if-statement (2)

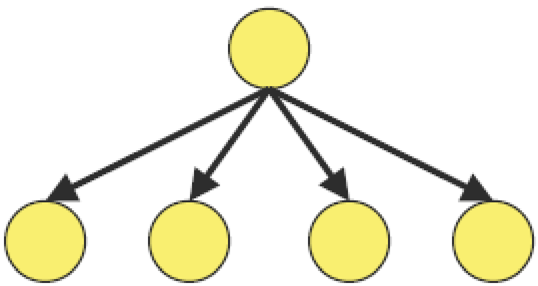
```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0)     -> n=n-2
:: (n % 3 == 0) -> n=3
:: else         -> skip
fi
```

- The **else** guard becomes executable if none of the other guards is executable.

give n a random value

```
if
:: skip -> n=0
:: skip -> n=1
:: skip -> n=2
:: skip -> n=3
fi
```

non-deterministic branching



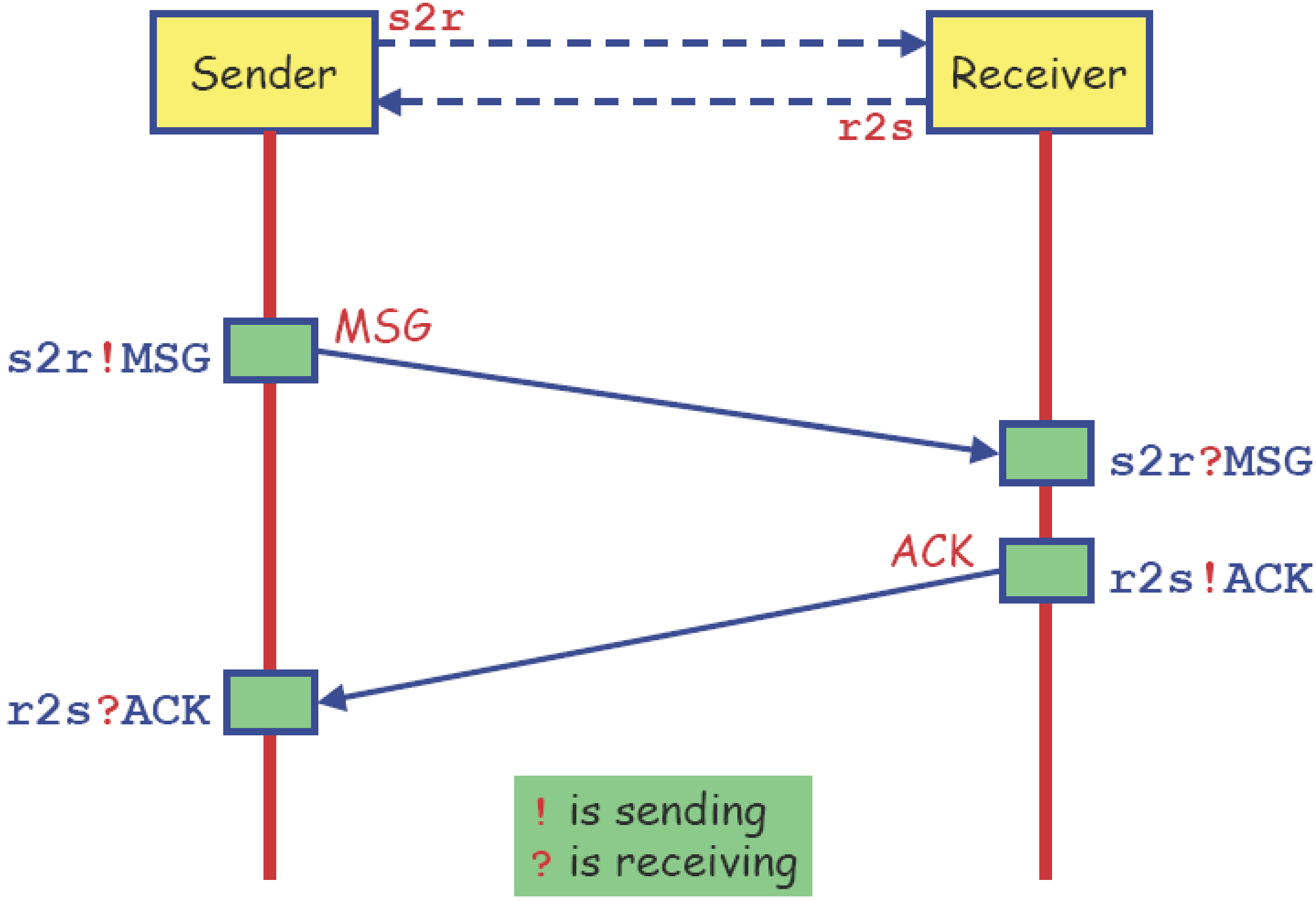
skips are redundant, because assignments are themselves always executable...

# do-statement (1)

```
do
:: choice₁ -> stat₁.₁; stat₁.₂; stat₁.₃; …
:: choice₂ -> stat₂.₁; stat₂.₂; stat₂.₃; …
:: …
:: choiceₙ -> statₙ.₁; statₙ.₂; statₙ.₃; …
od;
```

- With respect to the choices, a `do`-statement behaves in the same way as an **`if`**-statement.
- However, instead of ending the statement at the end of the choosen list of statements, a `do`-statement repeats the choice selection.
- The (always executable) **`break`** statement exits a **`do`**-loop statement and transfers control to the end of the loop.

# Communication (1)



# Communication (2)

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation (handshake)
- Both are defined as channels:

also called:
queue or buffer

```
chan <name> = [<dim>] of {<t1>,<t2>, ... <tn>};
```

name of
the channel

type of the elements that will be
transmitted over the channel

number of elements in the channel
dim==0 is special case: rendez-vous

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

array of
channels

# Communication (3)

- channel = FIFO-buffer (for `dim>0`)

! **Sending** - *putting a message into a channel*

`ch ! <expr`$_1$`>, <expr`$_2$`>, … <expr`$_n$`>;`

- The values of `<expr`$_i$`>` should correspond with the types of the channel declaration.
- A send-statement is executable if the channel is not full.

? **Receiving** - *getting a message out of a channel*

<var> + <const> can be mixed

`ch ? <var`$_1$`>, <var`$_2$`>, … <var`$_n$`>;` message passing

- If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the `<var`$_i$`>`s.

`ch ? <const`$_1$`>, <const`$_2$`>, … <const`$_n$`>;` message testing

- If the channel is not empty and the message at the front of the channel evaluates to the individual `<const`$_i$`>`, the statement is executable and the message is removed from the channel.

# Communication (4)

- Rendez-vous communication

  `<dim>` == 0

  The number of elements in the channel is now zero.

  – If send `ch!` is enabled and if there is a corresponding receive `ch?` that can be executed simultaneously and the constants match, then both statements are enabled.

  – Both statements will “handshake” and together take the transition.

- *Example:*

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

  – P wants to do `ch ! 1, 3+7`

  – Q wants to do `ch ? 1, x`

  – Then after the communication, `x` will have the value `10`.

# Promela syntax (1/5)

Declaration of variables:

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;
```

`turn` can assume values `0` or `1`.

`flag1` etc. can assume values `true` or `false`.

initial values: `0` and `false`, respectively

other data types: `byte` or others, e.g. records

# Promela syntax (2/5)

Declaration of processes:

```
active proctype p0() {
...
}
```

`proctype` defines a process type
(of which multiple instances may be activated at the same time)

`active` means that initially one instance of this process type is active.

One can also activate multiple instances initially, e.g. by

```
active [2] proctype my_process() {
...
}
```

# Promela syntax (3/5)

Jump labels / assignments / jumps:

```
again:  flag0 = true;
...
        goto again;
```

Empty statement:

```
skip
```

# Promela syntax (4/5)

Loops:

```
do
:: flag1 -> ...
:: else -> break;
od;
```

`flag1` and `else` are so-called guards.

Execution continues non-deterministically in a branch whose guard is true.

`else` branch can be executed iff no other guard is true.

`break` leaves the do loop.

# Promela syntax (5/5)

Branching:

```
if
:: turn == 1 -> ...
:: else -> ...;
fi
```

Syntax and semantics as in `do`, but only one single execution.

```
(turn != 1) -> ...
```

Guarded commands: Blocks execution until `turn` is not equal to `1`.

# Example: Mutual Exclusion algorithm by Dekker

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;

active proctype p0() {
...
}

active proctype p1() {
...
}
```

# Example: Contents of Process p0

```
active proctype p0() {
again:  flag0 = true;
        do
        :: flag1 ->
                if
                :: turn == 1 ->
                        flag0 = false;
                        (turn != 1) -> flag0 = true;
                :: else -> skip;
                fi
        :: else -> break;
        od;

        crit0 = true; /* critical section */ crit0 = false;

        turn = 1; flag0 = false;
        goto again;
}
```

Process p1: like p0, but swap 0 and 1

# Using Spin: Example

Let us use Spin to check whether Dekker's algorithm indeed fulfils the mutual exclusion property, i.e. both processes should not be in their critical sections at the same time:

$$\mathbf{G}\,\neg(\mathtt{crit0} \land \mathtt{crit1})$$

(interpreting the atomic propositions `crit0` and `crit1` naturally, i.e. they hold in states in which the respective boolean variables are true).

Spin syntax for the property: `[] !(crit0 && crit1)`

# Step-by-step instructions

Construct the promela model and save it to a file (e.g. `dekker.pml`).

Create a Büchi automaton (for the negated property) using Spin's `-f` option:

```
spin -f '![] !(crit0 && crit1)' > buechi
```

Construct the cross product (two steps):

- Build a C program (`pan`) that computes the product:

  ```
  spin -a -N buechi dekker.pml
  ```

- Compile the program: `make pan`

Check for emptiness (run the program): `./pan -a -n`

If there is an error, get the counterexample: `spin -p -t dekker.pml`

The course homepage has a script for all this: `spinLTL`

# Example: Checking properties

Checking the mutual exclusion property on the example (e.g. using

```
./spinLTL dekker.pml '[]!(crit0 && crit1)'
```

yields that the property holds.

Let us try another property: "Whenever p0 tries to enter its critical section, it should eventually succeed."

Spin syntax for the property: `[] (flag0 -> <> crit0)`

Checking the property using `spinLTL` yields a violation of the property!