Hello World

/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
printf("Hello process, my pid is: %d\n", _pid);
}
init {
  int lastpid;
  printf("init process, my pid is: %d\n", _pid);
  lastpid = run Hello();
  printf("last pid was: %d\n", lastpid);
}

Hello Processes

Hello()
print "Hello"

init()
print "init"
run Hello()
print "last"

Interleaving Semantics

- Promela processes execute concurrently.
- Non-deterministic scheduling of the processes.
- Processes are interleaved
  - Only one process can execute a statement at each point in time.
    - Exception: rendez-vous communication.
- All statements are atomic
  - Each statement is executed without interleaving it parts with other processes.
  - Each process may have several different possible actions enabled at each point of execution.
    - Only one choice is made, non-deterministically (randomly).
**Variables and Types (1)**

- Basic types
  - bit
  - byte
  - short
  - int
  - char

- Arrays
  - byte a[27];
  - byte array[4];

- Typedefs
  - typedef Record {
    short f1;
    byte f2;
  } Record;

**Statements (1)**

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

- An assignment is always executable.

- An expression is also a statement; it is executable if it evaluates to non-zero.
  
  \[
  2 < 3 \quad \text{always executable}
  \]
  
  \[
  x < 27 \quad \text{only executable if value of } x \text{ is smaller than 27}
  \]
  
  \[
  3 + x \quad \text{executable if } x \text{ is not equal to } -3
  \]

**Variables and Types (2)**

- Variables should be declared.

- Variables can be given a value by:
  - assignment
  - argument passing
  - message passing

- Variables can be used in expressions.

**Statements (2)**

- 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).

**Statements (3)**

- `assert(expr));
  - The `assert`-statement is always executable.
  - If `expr` evaluates to zero, SPIN will exit with an error, as the `assert` "has been violated".

- The `assert`-statement is often used within Promela models, to check whether certain properties are valid in a state.

```promela
proctype monitor() {
  assert(n <= 3);
}
proctype receiver() {
  ...
  toReceiver ? msg;
  assert(msg != ERROR);
  ...
}
```

**Mutual Exclusion (1)**

- `bit flag; /* signal entering/leaving the section */
- byte mutex; /* # procs in the critical section. */

```promela
proctype P(bit i) {
  [flag = 1] models while (flag == 1) /* wait */;
  mutex++; // C;/wait
  flag = 0;
}
proctype monitor() {
  assert(mutex != 2);
}
proctype monitor() {
  init {
    atomic { run P(0); run P(1); run monitor(); }
  }
}
```
### Mutual Exclusion (2)

```plaintext
byte n = 0;
mutex := mutex + 1;
x := 0;
if (mutex != 2):  
  mutex := mutex - 1;
  x := 1;
else: 
  mutex := mutex - 1;
  x := 1;

active proctype A() {  
    turn = B_TURN;
    x = 1;
    y = 0;
    assert (mutex != 2);
}

active proctype B() {  
    turn = A_TURN;
    x = 1;
    y = 0;
    assert (mutex != 2);
}

active proctype monitor() {  
    assert (mutex != 2);
}
```

Problem: invalid end state!
Both processes can pass execute `x = 1` and `y = 1` at the same time, and will then be waiting for each other.

### Mutual Exclusion (3)

```plaintext
first "software-only" solution to the Mutex problem (for two processes)
```

### Mutual Exclusion (4)

```plaintext
byte turn[2]; /* who's turn is it? */
byte mutex; /* # procs in critical section */

proctype P(bit i) {  
  if (i == A_TURN):  
      turn[i] = turn[1-i] + 1;
      mutex++;
      turn[i] = 0;
  else:  
      turn[i] = 0;
  
  P(i);
}

proctype monitor() {  
    assert (mutex != 2);
}
i init { atomic (run P(0); run P(1); run monitor());
```

More mutual exclusion algorithms
in (good-old) [Ben-Ari 1990]

#### if-statement (1)

```plaintext
if:  
  choice -> stat_1; stat_2; stat_3;  
  choice -> stat_1; stat_2; stat_3;  
  choice -> stat_1; stat_2; stat_3;  
fi;
```

- If there is at least one `choice` (guard) executable, the if-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no `choice` 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.

#### do-statement (1)

```plaintext
do:  
  choice -> stat_1; stat_2; stat_3;  
  choice -> stat_1; stat_2; stat_3;  
  choice -> stat_1; stat_2; stat_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 chosen 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.
do-statement (2)

- Example – modelling a traffic light

```promela
mtype = { RED, YELLOW, GREEN };
```

```promela
active proctype TrafficLight() {
    byte state = GREEN;
    do
        :: (state == GREEN) -> state = YELLOW;
        :: (state == YELLOW) -> state = RED;
        :: (state == RED) -> state = GREEN;
        od;
    Note: this do-loop does not contain any non-deterministic choice.
}
```

Communication in SPIN

- With more or less complexity each can implement the others
- Spin supports 1 and 4 (blocks send when buffer full), but with bounded buffers
- Buffer size = 0 \(\rightarrow\) synchronous communication
- Large buffer size approximates asynchronous communication

Communication (1)

- Synchronous broadcast
  - one sends, many receive synchronously
  - First variation: send never blocks process may receive if ready to ready
  - Second variation: send blocks until all possible recipients ready to receive

Communication (2)

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

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

- Both are defined as channels:
  - also called queue or buffer
  - name of the channel
  - type of the elements that will be transmitted over the channel
  - number of elements in the channel

Communication (3)

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

  ! Sending - putting a message into a channel
  ```promela
  ch ? <var>,<var>,...<var>;
  ```
  - The values of <var> 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
  ```promela
  ch ? <var>,<var>,...<var>;
  ```
  - If the channel is not empty, the message is fetched from the channel
  - The individual parts of the message are stored into the <var>.

```promela
<var> = <const> can be mixed
```

- If the channel is not empty and the message at the front of the channel evaluates to the individual <const>, the statement is executable and the message is removed from the channel.
Communication (4)

- Rendez-vous communication

<dia> == 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:

\[
\begin{align*}
\text{chan } \text{ch} & = \{0\} \text{ of } \{\text{bit, byte}\}; \\
P \text{ wants to do } & \text{ ch ! 1, 3+7} \\
Q \text{ wants to do } & \text{ ch ? 1, x} \\
\text{Then after the communication, } x \text{ will have the value 10.}
\end{align*}
\]

Alternating Bit Protocol (1)

- Alternating Bit Protocol
  - To every message, the sender adds a bit.
  - The receiver acknowledges each message by sending the received bit back.
  - To receiver only excepts messages with a bit that it excepted to receive.
  - If the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit.

Alternating Bit Protocol (2)

\[
\text{atomic } \{ \text{stat}_1, \text{stat}_2, \ldots, \text{stat}_n \}
\]

- can be used to group statements into an atomic sequence; all statements are executed in a single step (no interleaving with statements of other processes) is executable if \(\text{stat}_1\) is executable
- if a \(\text{stat}_i\) (with \(i>1\)) is blocked, the “atomicity token” is (temporarily) lost and other processes may do a step

(Hardware) solution to the mutual exclusion problem:

\[
\text{atomic } \{ \text{stat}_1; \text{stat}_2; \ldots; \text{stat}_n \}
\]

atomic

- can be used to group statements into an atomic sequence; all statements are executed in a single step (no interleaving with statements of other processes)
- is executable if \(\text{stat}_1\) is executable
- if a \(\text{stat}_i\) (with \(i>1\)) is blocked, the “atomicity token” is (temporarily) lost and other processes may do a step

(Hardware) solution to the mutual exclusion problem:

- To every message, the sender adds a bit.
- The receiver acknowledges each message by sending the received bit back.
- To receiver only excepts messages with a bit that it excepted to receive.
- If the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit.

atomic

D_step { stat1; stat2; ... ; statn }

- more efficient version of atomic; no intermediate states are generated and stored
- may only contain deterministic steps
- it is a run-time error if \(\text{stat}_i\) (i>1) blocks.

D_step is especially useful to perform intermediate computations in a single transition

atomic and D_step can be used to lower the number of states of the model
Checking for pure atomicity

- Suppose we want to check that none of the atomic clauses in our model are ever blocked (i.e. pure atomicity).

1. Add a global bit variable:
   ```
   bit aflag;
   ```

2. Change all atomic clauses to:
   ```
   atomic {
     assert(aflag=0);
     ...
     assert(aflag=1);
   }
   ```

3. Check that `aflag` is always 0.

   e.g. active process monitor {
       assert(!aflag);
   }

- Promela does not have real-time features.
  - In Promela we can only specify functional behaviour.
  - Most protocols, however, use timers or a timeout mechanism to resend messages or acknowledgements.

- timeout
  - SPIN’s timeout becomes executable if there is no other process in the system which is executable
  - so, timeout models a global timeout
  - timeout provides an escape from deadlock states
  - beware of statements that are always executable...

- goto label
  - each Promela statement might be labelled
  - quite useful in modelling communication protocols

```c
wait_ack: 
  if (B?ACK -> ab!1-ab ; goto success
  ; ChunkTimeout?SHAKE ->
  if (rc < MAX) -> rc++;
  if (i=1), (i=n), ab,d[i]
  goto wait_ack
  ; (rc >= MAX) -> goto error
fi
```
unless

{ <stats> } unless ( guard; <stats> )

- Statements in <stats> are executed until the first statement (guard) in the escape sequence becomes executable.
- resembles exception handling in languages like Java
- Example:

```c
proctype MicroProcessor() {
    ...
    /* execute normal instructions */
    unless ( port ? INTERRUPT; ... )
}
```

inline - poor man’s procedures

- Promela also has its own macro-expansion feature using the inline-construct.

```c
inline init_array(a) {
    d_step {
        i=0;
        do
            :: i<N -> a[i] = 0; i++
            :: else -> break
        od;
        i=0;
    }
    Be sure to reset temporary variables.
}
```

- error messages are more useful than when using #define
- cannot be used as expression
- all variables should be declared somewhere else