Hello World

```c
/* 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

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

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` (true/false)
  - `integer` (positive/negative)
  - `byte` (8-bit integer)
  - `int`, `short`, `long`, `double` (floating-point numbers)

- **Arrays**
  - `byte[27]`
  - `bitflags[8]`

- **Typedefs**
  - `typedef Record { short fi; byte li; } Record z; record declaration`

- **Execution**
  - Executable statements:
    - `2 < 3`
    - `x < 27`
    - `x + y` if `x` is not equal to `-3`
  - Blocked statements:
    - `2 < 3`
    - `x < 27`
    - `3 + x` if `x` is equal to `-3`

- **Expressions**
  - `assert(x < 3)` always executable
  - `if (x < 27)` only executable if value of `x` is smaller than 27
  - `x + y` executable if `x` is not equal to `-3`

**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.

**Variables and Types (2)**

- Variables should be declared.

- Variables can be given a value by:
  - assignment
  - argument passing
  - message passing (see communication)

- 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(x <= 3);
}
proctype receiver() {
    ... toReceive ? msg; assert(msg != ERROR);
    ...
}
```

**Mutual Exclusion (1)**

- The `mutex` and `monitor` statements are executed in a separate process.

- The `mutex` statement can only become executable if no other process is executing it.

- The `monitor` statement is always executable (but is not evaluated during verification, of course).

- The `assert` statement is always executable.

- The `printf` statement can only become executable if it is not evaluated during verification, of course.

- The `skip` statement is always executable.

**Demo**

- The `print` statement prints a message to the console.

- The `assert` statement checks whether a property holds.

- The `mutex` statement protects critical sections.

- The `monitor` statement allows processes to synchronize.

- The `printf` statement is not executed during verification.

- The `assert` statement is not evaluated during verification.

- The `skip` statement is always executable.

- The `printf` statement is executable if it is not evaluated during verification.
**Mutual Exclusion (2)**

```proctype P(bit i) {
  do
    turn[i] = 1 - turn[i];
    turn[i] = turn[i] + 1;
  od;
}
```

**Mutual Exclusion (3)**

```proctype A() {
  assert(mutex != 2);
  monitor() {
    turn[A_TURN] = 1;
    mutex--;
  }
}
```

**Bakery**

```proctype P(bit i) {
  do
    turn[i] = 1 - turn[i];
    turn[i] = turn[i] + 1;
  od;
}
```

**if-statement (1)**

```if
    choice -> stat2.1; stat2.2; stat2.3;
    ...;
    choice -> stat2.1; stat2.2; stat2.3;
    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)**

```do
    choice -> stat2.1; stat2.2; stat2.3;
    ...;
    choice -> stat2.1; stat2.2; stat2.3;
    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

```pascal
mtyp = { RED, YELLOW, GREEN } ;
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 (2)

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

- Both are defined as channels:

```pascal
chan <name> = [<dim>] of [type1:t1, type2:t2, ... typeN:tN];
```

*Note: name of the channel, type of the elements that will be transmitted over the channel, number of elements in the channel*  

- channel = FIFO-buffer (for \(\text{dim}>0\))

  ! Sending - putting a message into a channel
  ```
  ch ? <var1>, <var2>, ... <varN>;
  ```
  - The values of \(<\text{var}_1>, <\text{var}_2>, ... <\text{var}_N>\) 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
  ```
  ch ? <var1>, <var2>, ... <varN>;
  ```
  - If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the \(<\text{var}_1>, <\text{var}_2>, ... <\text{var}_N>\).
  - If the channel is not empty and the message at the front of the channel evaluates to the individual \(<\text{const}_1>, <\text{const}_2>, ... <\text{const}_N>\); message testing

```

Synchronous broadcast

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

Point-to-Point buffered message passing

- When buffer not full behaves like asynchronous
- When buffer full, two variations: block or drop message
- send never blocks
Communication (4)

- Rendez-vous communication
  
  \[\text{int} \text{init} \{ \text{run} \}
  \]

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

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 expected 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)

\[
\text{proctype} \text{P}(\text{bit} i) \{
\text{chan} \text{ch} = [0] \text{of} \{\text{bit, byte}\};
\text{chan} \text{toS} = [1] \text{of} \{\text{bit, byte}\};
\text{chan} \text{toR} = [2] \text{of} \{\text{bit, byte}\};
\text{proctype} \text{Sender}(\text{chan in}, \text{out}) \{
\text{bit} \text{sendbit}, \text{recvbit};
\text{do}\;
\text{::} \text{out} \text{MSG}, \text{sendbit} \rightarrow \text{in} \text{ACK,recvbit};
\text{if}\;
\text{::} \text{recvbit} \rightarrow \text{sendbit} = 1\text{-sendbit};
\text{else}
\text{::} \text{endif}
\text{od}
\}
\text{proctype} \text{Receiver}(\text{chan in, out}) \{
\text{bit} \text{recvbit};
\text{do}\;
\text{::} \text{in} \text{MSG}(\text{recvbit}) \rightarrow \text{out} \text{ACK}(\text{recvbit});
\text{od}
\}
\}
\]

- Atomic and \text{d_step} can be used to lower the number of states of the model

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

- 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 step

- (Hardware) solution to the mutual exclusion problem:

\[
\text{proctype} \text{P}(\text{bit} i) \{
\text{atomic} \{ \text{flag} = 1; \text{flag} = 1; \}
\text{mutex++;}
\text{mutex--;}
\text{flag} = 0;
\}
\]

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

- more efficient version of \text{atomic}; no intermediate states are generated and stored
- may only contain deterministic steps
- it is a run-time error if \text{stat}_1 (i>1) blocks.

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

- useful to perform intermediate computations in a single transition

- \text{atomic} and \text{d_step} can be used to lower the number of states of the model

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

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

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

- useful to perform intermediate computations in a single transition

\[
\text{atomic} \{ \text{stat}_1; \text{stat}_2; \ldots \text{stat}_n \}
\]
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 { stat, aflag=1; stat; ... stat, aflag=0; }
   ```

3. Check that aflag is always 0.

   ```
   []!aflag
   ```

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

   timeout (1)

- 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

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

```c
{ <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; ... }
}
```

`#define`LOSSY 1

```c
#ifdef LOSSY
active proctype Daemon() { /* steal messages */ }
#endif
```

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; 
    } 
}
```

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