Common LTL Predicates

LTL	Reads as...	Property
`[]p`	always p	invariance
`<>p`	eventually p	guarantee
`p -> <>q`	p implies eventually q	response
`p -> q U r`	p implies q until r	precedence
`[]<>p`	always eventually p	recurrence (progress)
`<>[]p`	eventually always p	stability (non-progress)
`<>p -> <>q`	eventually p implies eventually q	correlation

("The Spin Model-Checker", p137)

Model-Checking

- Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for that model, given an initial state.
- Due to initial work by Clarke and Emerson (1980, 1981, 1986).
- Model checking tools automatically check whether M ⊨ φ holds, where M is a (finite-state) model of a system and property φ is stated in some formal notation.
- SPIN is one of the most powerful model checkers
  - It uses Linear Temporal Logic
  - Its model M is the set of all possible paths from the starting state.

Promela

- Promela [*Pro*tocol/*Pro*cess *M*eta *La*nguage] is a modelling language, not a programming language.
- A Promela model consist of:
  - type declarations
  - channel declarations
  - global variable declarations
  - process declarations
  - [init process]

`mtype`, constants, typedefs (records)

- `simple vars`
- `structured vars`

behaviour of the processes: local variables + statements

initialises variables and starts processes

- A Promela model corresponds to a *finite* transition system, so:
  - no unbounded data / channels / processes / process creation

SPIN

- SPIN (Simple ProMeLa INterpreter) is a model-checking based verification tool for concurrent systems, e.g. concurrent programs.
- SPIN can also be used to ‘run’ the model, almost as if it was a program, to informally examine it.

Process (1)

- A process is defined by a **`proctype`** definition
- It executes concurrently with all other processes, independently of speed or behaviour
- A process communicates with other processes:
  - using global variables
  - using channels
- There may be several processes of the same type.
- Each process has its own local state:
  - the process counter (location within the proctype)
  - the contents of the local variables

Process (2)

- A process type (`proctype`) consists of
  - a name
  - a list of formal parameters
  - local variable declarations
  - the body

```
proctype Sender(byte in; byte out) {
    bit sndB;
    do
    :: out==10 ->
            in=4;
            if
            :: sndB == 1 -> sndB = 1-sndB
            :: else -> skip
            fi
    od
}
```

Hmmm, funny looking code up there ....

Process (3)

- Processes are created using the **`run`** statement (which returns the process id).
- Processes can be created at any point in the execution.
- Processes start executing after the **`run`** statement.
- Processes can also be created and start running by adding **`active`** in front of the **`proctype`** declaration.

Yes, that is an array of **`P`** ! **`P[0]`** and **`P[1]`**.

```
active [2] proctype P() {
    byte    i = 1;
    byte    temp;
    do
        :: ( i > TIMES ) -> break
        :: i > 10 ->
        temp = n;
        n = temp + 1;
        i++
    od;
    finished++;  /* Process terminates */
}
```

More funny looking code ....

SPIN Hello World Example

SPIN = *S*imple *P*romela *In*terpreter

```
/* 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);
    lastpid = run Hello();
    printf("last pid was: %d\n", lastpid);
}
```

```
$ spin -n2 hello.pr
          init process, my pid is: 1
          last pid was: 2
      Hello process, my pid is: 0
              Hello process, my pid is: 2
          last pid was: 3
                  Hello process, my pid is: 3
4 processes created
```

Promela Types

- Basic types
  - **`bit`** e.g. turn=1; range: [0..1]
  - **`bool`** e.g. flag; [0..1] or **`true`**, **`false`**
  - **`byte`** e.g. counter; [0..255]
  - **`short`** e.g. s; [-2^{15}.. $2^{15}-1$]
- Default initial value of basic variables (local and global) is 0.
- Most arithmetic, relational, and logical operators of C are supported, including bitshift operators.

Promela Types (2)

- Arrays
  - Zero-based indexing (one-dimensional only !)
- Records (“structs” in C/)
  - typedefs
- Mtypes
  - Promela supports one enumeration type called “**mtype**”.
  - It’s simply a set of names to be treated as distinct values

```
typedef Record {
  short f1;
  byte f2;
}
```

```
mtype = {name1, name2, … , nameN }
```

Special Variables

- Promela has special variables that make some internal properties visible.
  - These variables have names that start with an underscore ('_').
- Examples:
  - **`_pid`** return the process id of the current process
  - **`_nr_pr`** returns the number of currently active processes.

Statements

- A statement is either
  - *executable*: the statement can be executed immediately.
  - *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. E.g.
  - 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
  - "Executing" an expression has no effect on program variable state other than updating the process counter.

Statements (2)

- The skip statement is always executable.
  - it "does nothing", only changes the 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 ignored, i.e. not considered, during verification).

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

Let's demo the above!

Statements (3) — assert

- Format: **`assert(`**<expr>**`);`**
- The assert statement is always executable.
- If <expr> evaluates to zero, SPIN will exit with an error, as the *<expr> has been violated.*
- Often used within Promela models, to check whether certain properties are valid in a state.

sumofhellos.pml

- We can use Promela to model the **`sumofhellos.c`** C code used for Practical I
- We create a file called **`sumofhellos.pml`**
  - We use expression **`(_nr_pr == 1)`** to wait until all the **`PrintHello`** threads are done (i.e., until only the **`init`** process is still running).
- We can run a random simulation using SPIN on the command line:
  - **`$> spin sumofhellos.pml`**
- We see output similar to that of the C program.

Statements (4)

- All the statements mentioned so far (a.k.a. Basic statements) are ***atomic***:
  - when executable, once they execute, they do so atomically.
- Promela also has composite statements that compose other statements in some way
  - Sequential composition
  - Conditionals
  - Loops
- Composite statements are not atomic

Sequencing Statements

```
stat1 ; stat2 ; stat3 ; … ; statN
```

- We sequence statements simply by listing them, ***separated*** by semi-colons (‘*;*’).
- This is unlike in C, where ‘*;*’ is a statement *terminator*.
- Promela is liberal about semi-colons, so it will accept a last semicolon

```
stat1 ; stat2 ; stat3 ; … ; statN ;
```

or many !

```
stat1 ; stat2 ; stat3 ; … ; statN ;;;
```

If Statement

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

It's that weird code again!

- Each `::` introduces an alternative started by convention with a guard statement (expression)
  - if any guards are executable, then one is non-deterministically chosen.
  - Once an alternative has run, the conditional itself has terminated.
  - The optional else becomes executable if none of the other guards are executable.
- If no guard is executable, the if statement blocks, until least one becomes executable.
- The notation '->' is simply another way of writing ';'

Do Statement

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

A cunningly disguised while-loop!

- Similar to the If statement, but it repeats the choice at the end.
- Use the break statement to move on to the next statement after **`od`**.
- The strange notation derives from so-called Guarded Command Language (GCL)
  - Developed by Edsker Dijkstra to reason about program correctness
  - Its form is designed to emphasise any non-determinism present in a system.
  - The boolean expressions **`choice1, choice2`**,... are called the "guards".