Overview

- Basic building blocks of Promela programs.
- Structured data types.
- Process definition, instantiation & execution.
- Concurrency and Promela programs.
- Non-deterministic behaviour & basic synchronization.
Promela Programs

- The basic building blocks of Promela programs are:
  - processes
  - channels
  - variables

- Processes model the behaviour of components of a system and are by definition global objects.

- Channels and variables define the environment in which processes exist and can be either local or global.

Executability of Statements

- Promela does not make a distinction between a condition and a statement, e.g. the simple boolean condition \( a == b \) represents a statement in Promela.

- Promela statements are either executable or blocked. The execution of a statement is conditional on it not being blocked.

- Promela’s notion of statement executability provides the basic means by which process synchronization can be achieved.

\[
\begin{align*}
\text{while} \ (a != b) \ & \text{skip} /* \text{conventional busy wait} */ \\
(a == b) & \quad /* \text{Promela equivalent} */
\end{align*}
\]
**Variables and Basic Data Types**

- Promela variables provide the means of storing information about the system being modelled.
- A variable may hold global information on the system or information that is local to a particular component (process).
- Promela supports five basic data types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (bits)</th>
<th>Usage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>1</td>
<td>unsigned</td>
<td>0 \ldots 1</td>
</tr>
<tr>
<td>bool</td>
<td>1</td>
<td>unsigned</td>
<td>0 \ldots 1</td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>unsigned</td>
<td>0 \ldots 255</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>signed</td>
<td>(-2^{15} \ldots 2^{15} - 1)</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>signed</td>
<td>(-2^{31} \ldots 2^{31} - 1)</td>
</tr>
</tbody>
</table>

**Variable Declarations**

- Like all well-structured programming languages, Promela requires that variables must be declared before they can be used.
- Variable declarations follow the style of the C programming language, *i.e.* a basic data type followed by one or more identifiers and optional initializer:
  ```plaintext
  byte count, total = 0;
  ```
  An initializer must be an expression of the appropriate basic type.
- By default all variables of the basic types are initialized to 0. Note that as in C, 0 (zero) is interpreted as false while any non-zero value is interpreted as true.
Structured Data Types

- Arrays – an array type is declared as follows:
  ```
  int table[max]
  ```
  Note that this generates an array of \( \text{max}-1 \) integers, i.e.
  \( \text{table}[0], \text{table}[1], \ldots \text{table}[\text{max}-1] \)

- Enumerated Types – a set of symbolic constants is declared as follows:
  ```
  mtype = \{\text{LINE\_CLEAR}, \text{TRAIN\_ON\_LINE}, \text{LINE\_BLOCKED}\}
  ```
  Note: a program can only contain one \textit{mtype} declaration which
  must be global.

- Structures – a record data type is declared as follows:
  ```
  typedef msg \{byte data[4], byte checksum\}
  ```
  Note: Structure access is as in C:
  ```
  msg message; \ldots \text{message.data}[0]
  ```

Identifiers, Constants & Expressions

- Identifiers: An identifier is a single letter, a
  period symbol, or underscore followed by zero
  or more letters, digits, periods or underscores.

- Constants: A constant is a sequence of digits
  that represents a decimal integer. Symbolic
  constants can be defined by means of \textit{mtype}
  or via a C-style macro definition, \textit{e.g.}
  ```
  \#define MAX 999
  ```

- Expressions: An expression is built up from
  variables, identifiers and constants using the
  following operators:
  ```
  +, -, *, /, \%, --, ++, arith
  >, >=, <, <=, ==, !=, relational
  &&, ||, !,
  &\&, |, |
  &, |, \^, >>, <<, bits
  !, ?,
  () , [] ,
  ```
  group/index
Process Types

- A process declaration begins with the keyword `proctype` and contains:
  - process identifier;
  - formal parameter list;
  - sequence of local variable declarations & statements
- Syntactically a process declaration has the following form:
  ```
  proctype name( /* formal parameter list */ )
  {
   /* local declarations and statements */
  }
  ```
  Note: /* and */ delimit comments in Promela.

The init Process

- All Promela programs must contain an `init` process which is similar to the `main()` function within a C program, i.e. the execution of a Promela program begins with the `init` process.
- An `init` process takes the form:
  ```
  init { /* local declarations and statements */ }
  ```
  The simplest and may be one of the least useful of Promela programs takes the form:
  ```
  init { skip }
  ```
  Note: `skip` denotes the null statement.
- While a `proctype` definition declares the behaviour of a process, the instantiation and execution of a process definition is co-ordinated via the `init` process.
Hello World

- A simple two process system:
  proctype hello(){ printf("Hello") }
  proctype world(){ printf("World\n") }
  init { run hello(); run world () }
- The init process initiates the execution of an instance of the hello process and the world process.
- The run operator is executable only if process instantiation is possible. A process instantiation may be blocked if too many processes are already running.
- If a run is executable then a pid or run-time process identification number is returned. The pid for a process can be accessed via the predefined local variable _pid.
- The execution of run does not wait for the associated process to terminate, i.e. further applications of run will be executed concurrently.

Active Proctypes

- A refinement to our simple two process system:
  active proctype hello(){ printf("Hello") }
  active proctype world(){ printf("World\n") }
- The keyword active can prefix any proctype declaration.
- The effect of active is to create an instance of the associated proctype within the initial system state.
- Multiple instances of the same proctype declaration can be generated using an optional array suffix, e.g.
  active [4] proctype hello(){ printf("Hello") }
  active [7] proctype world(){ printf("World\n") }
  Note: the above will generate 4 instances of hello and 7 instances of world.
Processes as Automata

byte x = 2, y = 3;
proctype A(){x = x + 1}
proctype B(){x = x - 1; y = y + x}

Concurrency via Interleaving

init { run A(); run B() }
Deterministic & Non-Deterministic Behaviour

- **Deterministic behaviour:** a process is deterministic if for a given start state it behaves in exactly the same way if supplied with the same stimuli from its environment.

- **Non-deterministic behaviour:** a process is non-deterministic if it need not always behave in exactly the same way each time it executes from a given start state with the same stimuli from its environment.

More Non-Deterministic Behaviour

- Consider the following two process system:

```plaintext
byte state = 1;
proctype A() { (state == 1) -> state = state + 1 }
proctype B() { (state == 1) -> state = state - 1 }
init { run A(); run B() }
```

Note that $S1 \rightarrow S2$ and $S1; S2$ are equivalent.

- Note that if process A (B) terminates before process B (A) begins execution then B (A) will be blocked forever on the initial condition, i.e. $(state == 1)$.

- Note that if both A and B pass the condition simultaneously then both processes can terminate but the final value of `state` is unpredictable, i.e. 0, 1 or 2.
# Dekker’s Solution

```c
#define true 1
#define false 0
#define Aturn false
#define Bturn true
bool Aruns, Bruns, t;

proctype A()
{
    Aruns = true;
    t = Bturn;
    (Bruns == false || t == Aturn); /* S1 */
    /* critical section */
    Aruns = false
}

proctype B()
{
    Bruns = true;
    t = Aturn;
    (Aruns == false || t == Bturn); /* S2 */
    /* critical section */
    Bruns = false
}

init { run A(); run B() }
```

## Observations on Dekker’s Solution

- Statements S1 and S2 ensure synchronization between A and B on **critical section** access.

- Consider the case of S1 which occurs within the definition of process A: (Bruns == false || t == Aturn)
  - If the left disjunct holds then process B is not executing so it is safe for A to enter the critical section.
  - Else if the right disjunct holds t must be false so it is safe for A to enter the critical section because both disjuncts of S2 must be false, i.e. process B is blocked.
  - Else process A is blocked. However, the right disjunct of S2 must hold so it is safe for B to enter the critical section.
  - On exiting the critical section process A sets Aruns to false which has the effect of unblocking process B.
Atomic Sequences

- Promela provides another means of avoiding the undesirable interleaving problem illustrated above via the atomic operator.
- Consider the following refinement to the two process system:

```plaintext
byte state = 1;
proctype A()
{ atomic{ (state == 1) -> state = state + 1 } }
proctype B()
{ atomic{ (state == 1) -> state = state - 1 } }
init { run A(); run B() }
```

- The final value of the global variable state will be either 2 or 0, depending upon which process executes.
- Note that an atomic sequence restricts the level of interleaving so reduces the complexity when it comes to validating a Promela model.

Summary

Learning outcomes:
- To be able to understand and construction simple Promela programs exploiting both local and global data objects;
- To understand the Promela model for concurrent process execution;
- To be able to model synchronous behaviour between processes;

Recommended reading:
- “Concise Promela Reference” — see course homepage
- “Basic Spin Manual” — see course homepage