Distributed Systems Programming F29NM1

Promela II

Andrew Ireland

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Overview

- Control flow constructs;
- Channel based process communication;
- Procedures and recursion.

Control Flow

- In the "Promela I" lecture three ways for achieving control flow were introduced:
  - Statement sequencing;
  - Atomic sequencing;
  - Concurrent process execution.
- Promela supports three additional control flow constructs:
  - Case selection
  - Repetition
  - Unconditional jumps

Case Selection

- What follows is an example of **case selection** involving two statement sequences:

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

  Note that each statement sequence is prefixed by a `::`. The executability of the first statement (**guard**) in each sequence determines sequence is executed.
- Guards need not be mutually exclusive:

  ```
  if
  :: (x >= y) -> max = x;
  :: (y >= x) -> max = y;
  fi
  ```

  Note: if `x` and `y` are equal then the selection of which statement sequence is executed is decided at random, giving rise to non-deterministic choice.

Repetition

- What follows is an example of **repetition** involving two statement sequences:

```
do
:: (x >= y) -> x = x - y; q = q + 1;
:: (y > x)  -> break;
od
```

  Note that the first statement sequence denotes the body of the loop while the second denotes the termination condition.

- Termination, however, is not always a desirable property of a system, in particular, when dealing with reactive systems:

```
do
:: (level > max) -> outlet = open;
:: (level < min) -> outlet = close;
od
```


Unconditional Jumps

- Promela supports the notion of an unconditional jump via the `goto` statement.
- Consider the following refinement of the division program given above:

```
      do
      :: (x >= y) -> x = x - y; q = q + 1;
      :: (y > x)  -> goto done;
      od;
done:
      skip
```

  Note that `done` denotes a label. A label can only appear before a statement. Note also that a `goto`, like a `skip`, is always executable.

Timeouts

- Reactive systems typically require a means of aborting/rebooting when a system deadlocks. Promela provides a primitive statement called `timeout` which enables such a feature to be modelled.
- To illustrate, consider the following process definition:

```
proctype watchdog ()
{
   do
   :: timeout -> guard!reset
   od
}
```

  The `timeout` condition becomes true when no other statements within the overall system being modelled are executable.

Exceptions

- Another useful exception handling feature is supported by the `unless` statement which takes the following general form:

```
{ statements-1 } unless { statements-2 }
```

  Execution begins with `statements-1`. Before execution of each statement the executability of the first statement within `statements-2` is checked. If the first statement is executable then control is passed to `statements-2`. If however the execution of `statements-1` terminates successfully then `statements-2` is ignored.
- Consider an alternative `watchdog` process:

```
proctype watchdog ()
{ do
  process_data() unless guard?reset; process_reset()
  od
}
```

Message Channels

- So far global variables have provided the only means of achieving communication between distinct processes.
- However, Promela supports **message channels** which provide a more natural and sophisticated means of modelling inter-process communication (data transfer).
- A channel can be defined to be either local or global. An example of a channel declaration is:

  ```
  chan in_data = [8] of { byte }
  ```

  which declares a channel that can store up to 8 messages of type `byte`.
- Multiple field messages are also possible:

  ```
  chan out_data = [8] of { byte, bool, chan }
  ```


Sending Messages

- Sending messages is achieved by the `!` operator, *e.g.*

  ```
  in_data ! 4;
  ```

  This has the effect of appending the value `4` onto the end of the `in_data` channel.
- If multiple data values are to be transferred via each message then commas are used to separate the values, *e.g.*

  ```
  out_data ! x + 1, true, in_data;
  ```

  where `x` is of type `byte`.
- Note that the executability of a send statement is dependent upon the associated channel being non-full, *e.g.* the following statement will be blocked:

  ```
  in_data ! 4;
  ```

  unless `in_data` contains at least one empty location.

Receiving Messages

- Receiving messages is achieved by the `?` operator, *e.g.*

```
in_data ? msg;
```

This has the effect of retrieving the first message (FIFO) within the `in_data` channel and assigning it to the variable `msg`.

- If multiple data values are to be transferred via each message then commas are used to separate the values, *e.g.*

```
out_data ? value1, value2, value3;
```

- Note that the executability of a receive statement is dependent upon the associated channel being non-empty, *e.g.* the following statement will be blocked:

```
in_data ? value;
```

unless `in_data` contains at least one message.

Some Observations & Notations

- If more data values are sent per message than can be stored by a channel then the extra data values are lost, *e.g.*

```
in_data ! msg1, msg2;
```

here the `msg2` will be lost.

- If fewer data values are sent per message than are expected then the missing data values are undefined, *e.g.*

```
out_data ! 4, true;
out_data ? x, y, z;
```

here `x` and `y` will be assigned the values `4` and `true` respectively while the value of `z` will be undefined.

- Alternative (& equivalent) notations:

```
out_data!exp1,exp2,exp3;   out_data!exp1(exp2,exp3);
out_data?var1,var2,var3;   out_data?var1(var2,var3);
```

Additional Channel Operations

- Determining the number of messages in a channel is achieved by the `len` operator, *e.g.*

  ```
  len(in_data)
  ```

  If the channel is empty then the statement will block.

- The `empty`, `full` operators determine whether or not messages can be received or sent respectively, *e.g.*

  ```
  empty(in_data);          full(in_data)
  ```

- Non-destructive retrieve:

  ```
  out_data ? [x, y, z]
  ```

  Returns 1 if `out_data?x,y,z` is executable otherwise 0. No side-effects – evaluation, not execution, *i.e.* no message retrieved.

Channels as Parameters

- Consider the following:

```
proctype A(chan q1)
{    chan q2;
     q1?q2; q2!99
}
proctype B(chan qforb)
{    int x;
     qforb?x; x++;
     printf("x == %d\n", x)
}
init {chan qname = [1] of { chan };
      chan qforb = [1] of { int };
      run A(qname); run B(qforb);
      qname!qforb
     }
```

- What will be the side-effect of running this program?

Rendez-Vous Communication

- Our discussion of message channels so far has implicitly focussed upon **asynchronous** communication between processes, *e.g.*

  ```
  chan name = [N] of { byte }
  ```

  where `N` is a positive constant that defines the number of locations allocated to the channel.

- However, **synchronous** communication between processes can be achieved by setting `N` to be `0`, *e.g.*

  ```
  chan name = [0] of { byte }
  ```

  This is known as a **rendezvous**, a channel where a message can be passed but not stored, *e.g.* `name!2` is blocked until a corresponding `name?msg` is executable.

- Note: rendezvous communication is binary.

A Rendez-Vous Example

- Consider the following:

  ```
  #define msgtype 33
  chan name = [0] of { byte, byte }
  proctype A()
  {    name!msgtype(124); name!msgtype(121) }
  proctype B()
  {    byte state;
       name?msgtype(state)
  }
  init { atomic { run A(); run B() }}
  ```

- Channel `name` is a global rendezvous. Both `A` and `B` will synchronous on their first statements. The effect will be to transfer the value `124` from `A` to the local variable `state` within `B`. Further execution is blocked because the second `send` within `A` has no matching `receive` within `B`.

Dijkstra's Semaphores

```
#define p 0
#define v 1

chan sema = [0] of { bit };
proctype semaphore()
{       do
        :: sema!p -> sema?v
        od
}
proctype user()
{       sema?p;
        /* critical section */
        sema!v;
        /* non-critical section */
        skip
}

init
{       atomic {
                run semaphore();
                run user();
                run user();
                run user()
               }
}
```


Procedures & Recursion

- Integer division revisited:

```
proctype division(int x,y,q; chan res)
{
   if
   :: (y > x)  -> res!q,x;
   :: (x >= y) -> run division(x - y, y, q + 1, res);
   fi
}

init{ int q,r;
      chan child = [1] of { int, int };
      run division(7, 3, 0, child);
      child ? q,r;
      printf("result: %d %d\n", q,r)
}
```

- Note that the algorithm is tail-recursive, *i.e.* the final result is communicated back to `init` directly.

Procedures & Recursion

- An non tail-recursive algorithm:

```
proctype fact(int n; chan res)
{    int result;
     if
     :: (n <= 1) -> res!1;
     :: (n >= 2) -> chan child = [1] of { int };
                    run fact(n - 1, child);
                    child ? result; res!n * result
     fi
}
init{ int result;
      chan child = [1] of { int };
      run fact(5, child); child ? result;
      printf("result: %d\n", result)}
```

- Note that each recursive call results in the dynamic creation of a child process. A process does not terminate until all its child processes terminate.

Summary

Learning outcomes:

- To be able to understand and construct simple programs exploiting Promela's control flow constructs, including `timeout` and `unless`.
- To be able to understand and construct asynchronous and synchronous behaviour between processes using message channels;
- To be able to use Promela to model procedures and recursion.

Recommended reading:

- "Concise Promela Reference" — see course homepage
- "Basic Spin Manual" — see course homepage