Contents

Message Channels

- Channels are used to transfer messages between active processes.
- They store messages in first-in first-out order.
- Two types:
  - buffered channels,
  - rendezvous ports, also called synchronous channels.

Buffered Channels

- Declaration:

  ```
  chan qname = [16] of { short, byte, bool }
  ```

  This channel can store up to 16 messages, each consisting of 3 fields of the types listed.

Buffered Channels

- Declaration:

  ```
  chan qname = [16] of { short, byte, bool }
  ```

  This channel can store up to 16 messages, each consisting of 3 fields of the types listed.
- A field can be of a pre-defined or user-defined type, but not an array (but a typedef can contain an array!).

Buffered Channels

- Declaration:

  ```
  chan qname = [16] of { short, byte, bool }
  ```

  This channel can store up to 16 messages, each consisting of 3 fields of the types listed.
- A field can be of a pre-defined or user-defined type, but not an array (but a typedef can contain an array!).
- Sending a message:

  ```
  qname!expr1,expr2,expr3
  ```

  The process blocks if the channel is full.

Buffered Channels

- Declaration:

  `chan qname = [16] of { short, byte, bool }`

  This channel can store up to 16 messages, each consisting of 3 fields of the types listed.
- A field can be of a pre-defined or user-defined type, but not an array (but a typedef can contain an array!).
- Sending a message:

  `qname!expr1,expr2,expr3`

  The process blocks if the channel is full.
- Receiving a message:

  `qname?var1,var2,var3`

  The process blocks if the channel is empty.

Buffered Channels

- Declaration:

  `chan qname = [16] of { short, byte, bool }`

  This channel can store up to 16 messages, each consisting of 3 fields of the types listed.
- A field can be of a pre-defined or user-defined type, but not an array (but a typedef can contain an array!).
- Sending a message:

  `qname!expr1,expr2,expr3`

  The process blocks if the channel is full.
- Receiving a message:

  `qname?var1,var2,var3`

  The process blocks if the channel is empty.
- Useful pre-defined functions `len`, `empty`, `nempty`, `full`, `nfull`:

  `len(qname)`

Alternative

- The first message field is a message type indication:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

Alternative

- The first message field is a message type indication:

  `qname!expr1(expr2,expr3)`
  `qname?var1(var2,var3)`

- Some parameters can be given as constants:

  `qname?cons1,var2,cons2`

  The process blocks if the channel is empty or if the sent values do not match the constants.

Rendezvous Ports

- Declaration of a rendezvous port (it pass single byte messages)

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

Rendezvous Ports

- Declaration of a rendezvous port (it pass single byte messages)
  `chan port = [0] of { byte }`
- The channel size is zero:
  the channel port can pass, but can not store messages!

Rendezvous Ports

- Declaration of a rendezvous port (it pass single byte messages)
  `chan port = [0] of { byte }`
- The channel size is zero:
  the channel port can pass, but can not store messages!
- Message interaction is synchronous: two processes execute a send and a receive statement at the same time (as a single atomic operation).

```
mtype = { msgtype };
chan name = [0] of { mtype, byte };
active proctype A()
{       name!msgtype(124);
        name!msgtype(121)
}
active proctype B()
{       byte state;
        name?msgtype(state)
}
```

Channels of channels

- Message parameters are always passed by value.
- We can pass the value of a channel from a process to another.

```
mtype = { msgtype };
chan glob = [0] of { chan };
active proctype A()
{       chan loc = [0] of { mtype, byte };
        glob!loc;
        loc?msgtype(121)
}
active proctype B()
{       chan who;
        glob?who;
        who!msgtype(121)
}
```

Contents

Statements

- Every statement is either *executable* or *blocked*.
- Three main types of statements:
  - print statements
  - assignments
  - expression statements
- Print statements and assignments are always executable (as well as `skip`, `assert`, ...).
- Expression statements are executable iff they evaluate to true.
  - `(2 < 3)` always executable;
  - `(x < 27)` blocked until `x` is less than 27;
  - `(3 + x)` executable when `x` differs from `-3`.
- Expressions must be side effect free (e.g. `b = c++` is not valid).
- Exception: the `run` statement can be considered as a blocking expression:
  - it blocks when there are 255 processes alive;
  - if it does not block, it creates a new process.

Contents

Basic verification 1

Given the following PROMELA program:

```
active proctype P() {
  int x = 0;
  x++;
  int y = x;
  assert(y == 1);
}
```

Is the assertion invalid? If yes, why?

Basic verification 1

Given the following PROMELA program:

```
active proctype P() {
  int x = 0;
  x++;
  int y = x;
  assert(y == 1);
}
```

Is the assertion invalid? If yes, why?

All variable declarations are always implicitly moved to the beginning of process.

Basic verification 2

```
chan com = [0] of { byte };
byte value;
bool d;
proctype p() {
  byte i;
  do
  :: if
     :: i >= 5 -> break
     :: else -> printf("Doing something else\n"); i ++
     fi
  :: com ? value; printf("p received: %d\n",value)
  od;
  d = 1
}
init {
     atomic {
        run p();
     }
     end: com ! 100;
}
```

Is it possible that process p does not read from the channel at all?

Basic verification 2

```
chan com = [0] of { byte };
byte value;
bool d;
proctype p() {
  byte i;
  do
  :: if
     :: i >= 5 -> break
     :: else -> printf("Doing something else\n"); i ++
     fi
  :: com ? value; printf("p received: %d\n",value)
  od;
  d = 1
}
init {
     atomic {
        run p();
     }
     end: com ! 100;
}
```

Is it possible that process p does not read from the channel at all? Yes

Sum of array elements

Write a PROMELA model for summing up an array of integers.

- Declare and (nondeterministically) initialize an integer array.
- Add a loop that sums up the elements.

Channels

Declare a rendezvous channel and create two processes:

- The first process sends the numbers 0 through 9 onto the channel.
- The second process reads the values of the channel and outputs them.

Channels

Declare a rendezvous channel and create two processes:

- The first process sends the numbers 0 through 9 onto the channel.
- The second process reads the values of the channel and outputs them.

Replace the rendezvous with a buffered channel and check how the behavior changes.