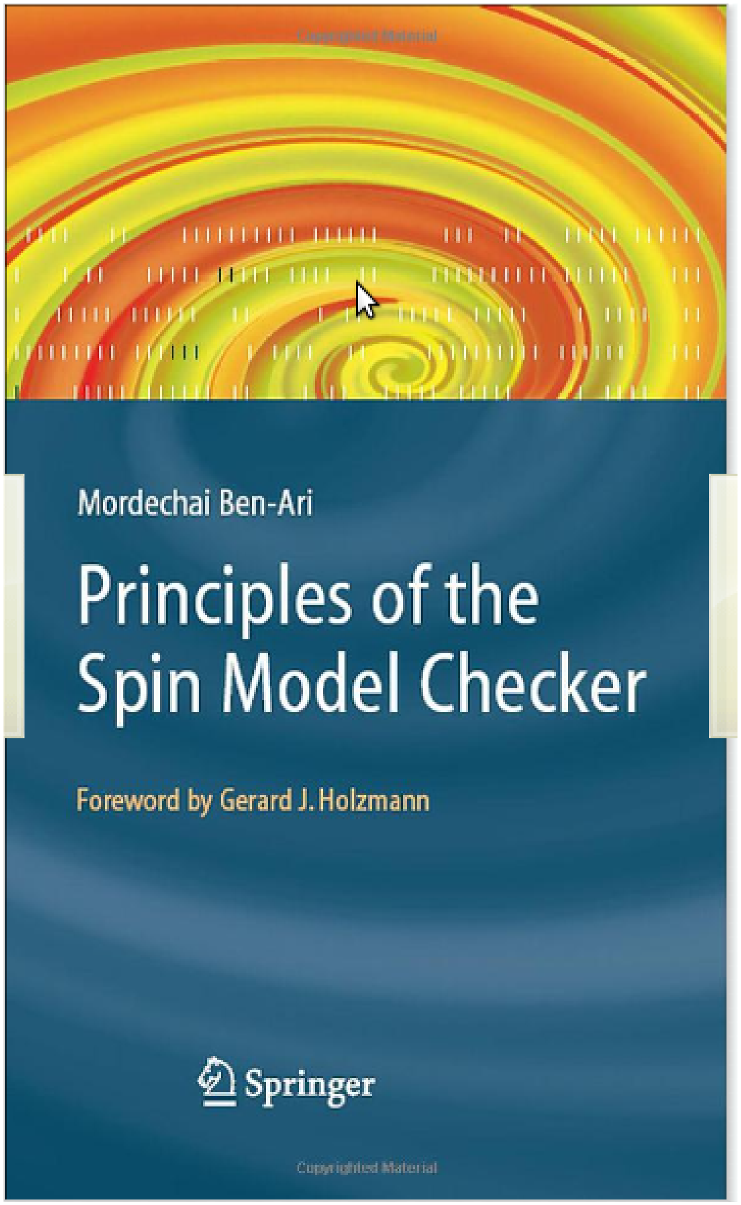

Mordechai Ben-Ari

Principles of the
Spin Model Checker

Foreword by Gerard J. Holzmann

Springer

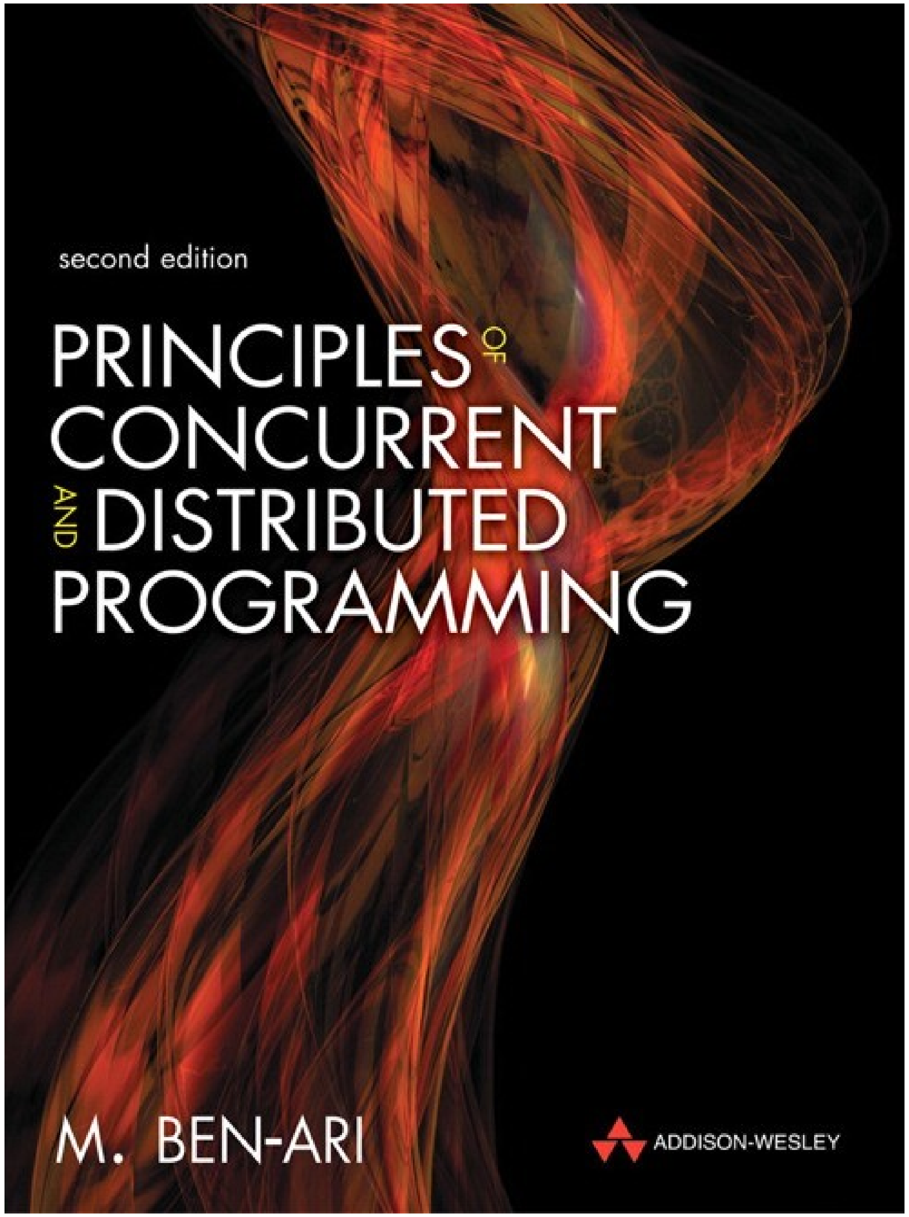
Copyrighted Material

Mordechai Ben-Ari

Principles of the Spin Model Checker

Springer, 2008

ISBN: 978-1-84628-769-5



Mordechai Ben-Ari

Principles of Concurrent and Distributed Programming

Second Edition
Addison-Wesley, 2008

ISBN: 978-0-32131-283-9

http://www.springer.com/computer/swe/book/978-1-84628-769-5

Supplementary material (zip, 38 kB)

978-1-84628-769-5-additional material.zip

Chapter 7

Channels

Distributed systems are computer systems consisting of a set of *nodes* connected by *communications channels*.

The most familiar distributed system is, of course, the Internet, which consists of millions of computer connected by communications networks implemented with wires, optical fibers and microwave radio.

Protocols such as TCP/IP and HTTP define how data are moved between nodes of the network.

The network itself is quite complex, using computers to perform essencial communications functions such as routing, name lookup, and error correction.

To model a distributed system we abstract away details of the network and its protocols, and model nodes as concurrent processes and communications networks as channels over which processes can send and receive messages.

The most widely used formalism for modeling distributed systems is called *Communicating Sequential Processes (CSP)*, after a 1978 article by that name, written by C.A.R. Hoare.

CSP was the inspiration for the communications constructs in several programming languages such as OCCAM and Ada, as well as for the channel construct in Promela.

Warning

In CSP and OCCAM a channel is always associated with a pair of processes; that is, exactly one process can send to each channel and exactly one process can receive from a channel.

In Promela channels are global entities not associated with processes, so any process can send a message on any channel and receive a message from any channel.

In fact, a process can send messages to and receive messages from a single channel!

Throughout this chapter we will use a *client-server system* as a running example.

A number of processes called *clients* send *requests* to other processes called *servers*.

A server performs a service and can return a *result* to a client.

Section 7.1

Channels in Promela

A *channel* in Promela is a data type with two operations, *send* and *receive*.

Each channel has associated with it a *message type*; once a channel has been initialized, it can only send and receive messages of its message type.

At most 255 channels can be created.

Listing 7.1. Client-server using channels (`rendezvous1.pml`)

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

active proctype Server() {
  byte client
end:
  do
  :: request ? client
       printf("Client %d\n", client)
  od
}

active proctype Client0() {
  request ! 0
}

active proctype Client1() {
  request ! 1
}
```

Listing 7.1 shows a model of a client-server system, where two clients are connected to a single server through a channel called `request`. The channel is declared with an *initializer* specifying the *channel capacity* and the message type:

```
    chan ch = [capacity] of { typename, ..., typename }
```

The channel capacity must be a nonnegative integer constant. The message type specifies the structure of each message that can be sent on the channel as a sequence of fields; the number of fields and the type of each field are specified in the declaration.

In the program in Listing 7.1, the capacity of the channel is zero, while the message type consists of a single field of type **`byte`**.

There are two types of channels with different semantics: *rendezvous* channels of capacity zero and *buffered channels* of capacity greater than zero.

Warning

The type of a message field cannot be an array; however, the type can be a **`typedef`** and the **`typedef`** can contain an array.

Syntactically, the *send statement* consists of a channel variable followed by an exclamation point and then a sequence of *expressions* whose number and types match the message type of the channel.

The *receive statement* consists of a channel variable followed by question mark and a sequence of *variables*.

Semantically, the expressions in the send statement are evaluated and their values are transferred through the channel; the receive statement assigns these values to the variables listed in the statement.

In the program in Listing 7.1, each client sents an integer value on the channel (lines 13, 17); the server receives the values and assigns them to the variable `client` (line 7).

Clearly, a receive statement cannot be executable unless a message is available in the channel. Receive statements will frequently appear as guards in an **`if`**- or **`do`**-statement, as shown in line 7 of Listing 7.1.

Note the use of the `end` label in the server; while it is reasonable for client processes to send a number of requests and then terminate, a server process never knows when it will be called upon to process a request, so it should never terminate. The label ensures that an end state with the server blocked on a receive statement is not considered invalid.

A bit of syntactic sugar for send and receive statements is supported: the list of expressions `ch!e1,e2,...` can be written: `ch!e1(e2,...)`. This is primarily used when the first arguments is an **`mtype`**, indicating the type of the message. For example, given the declarations:

```
mtype { open, close, reset }
chan ch = [1] of { mtype, byte, byte }
byte id, n
```

a send statement can be written in either of the following formats:

```
ch ! open, id, n
ch ! open(id, n)
```

Subsection 7.1.1

Channels and channel variables

The type of all channels is **chan** and a channel variable holds a reference or "handle" to the channel itself, which is created by an initializer. This means that channel variables can appear in assignment statements or, more commonly, as parameters to a **proctype** or **inline**:

```
chan ch1 = [0] of { byte }
chan ch2 = [0] of { byte, byte }
proctype P(chan c) {
  c ! 5
}
init {
  run P(ch1)
  run P(ch2)
}
```

Since the message in a send statement must match the message type of the channel, the send in the second instantiation of P causes a runtime, but not a compile-time, error, demonstrating that the channel *variable* is not typed with a message type.

Section 7.1.1. Channels and channel variables

```
$ cat -n listing_7_1_1.pml | expand
     1  chan ch1 = [1] of { byte }
     2  chan ch2 = [1] of { byte, byte }
     3
     4  proctype P(chan c) {
     5      byte x, y
     6      c ! 5, 7
     7      c ? x, y
     8      printf("x=%d, y=%d\n", x, y)
     9  }
    10
    11  init {
    12      run P(ch1)
    13      run P(ch2)
    14  }

$ spin listing_7_1_1.pml | expand
          x=5, y=0
              x=5, y=7
3 processes created
```

Advanced: Local channels

Channels are usually initialized globally, though one can be declared and initialized locally and then passed to another process in a message. However, if a channel is declared and initialized within a process and the process then dies, the channel is no longer accessible.

Section 7.2

Rendezvous channels

Listing 7.2. Simple program with rendezvous (`rendezvous0.pml`)

```
mtype { red, yellow, green }
chan ch = [0] of { mtype, byte, bool }

active proctype Sender() {
  ch ! red, 20, false
  printf("Sent message\n")
}

active proctype Receiver() {
  mtype color
  byte time
  bool flash
  ch ? color, time, flash
  printf("Received message %e, %d, %d\n",
         color, time, flash)
}
```

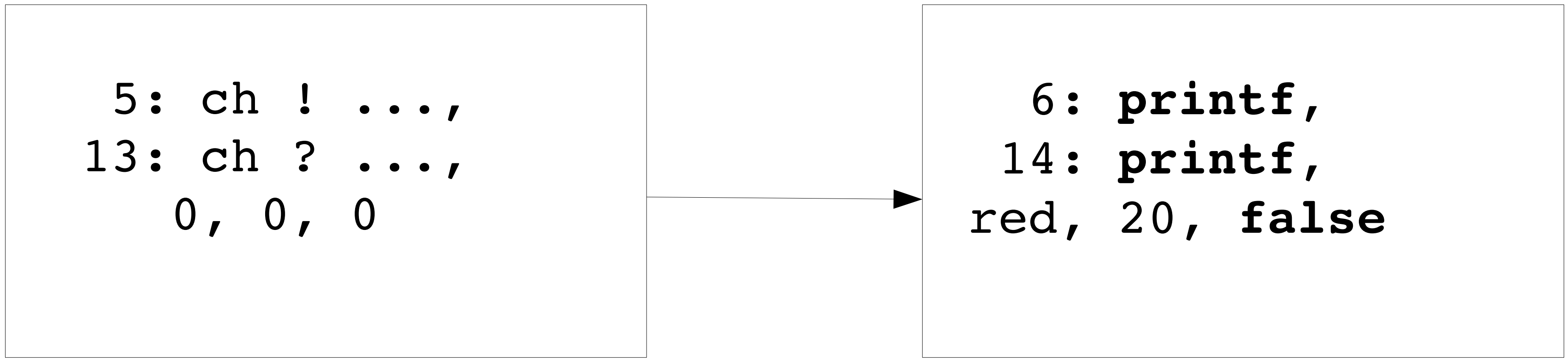
```
$ spin listing_7_2.pml
      Sent message
          Received message red, 20, 0
2 processes created
```

A channel declared with a capacity of zero is a *rendezvous channel*. This means that the transfer of the message from the *sender* (a process with a send statement) to the *receiver* (a process with a receive statement) is *synchronous* and is executed as a single atomic operation.

For the program in Listing 7.2, the atomic transfer is suggested by the arrow in the following diagram that goes directly from the send statement to the receive statement, so that there is no state between sending and receiving:

$$\begin{tikzcd}
\texttt{Sender} & \texttt{Receiver} \\
\vdots & \vdots \\
\texttt{(green,20,}\mathbf{\texttt{false}}\texttt{)} \arrow[r] & \texttt{(color,time,flash)} \\
\vdots & \vdots
\end{tikzcd}$$

When the location counter of the sender is at the send statement (line 5), it is said to *offer* to engage in a rendezvous. If the location counter of the receiver is at the matching receive statement (line 13), the rendezvous can be *accepted* and the values of the data in the send statement are copied to the variables in the receive statement. The state change is shown in the following diagram:



The rendezvous is one *atomic* operation; even if there were other processes, no interleaving could take place between the execution of the send statement and the receive statement.

A send statement that offers to engage in a rendezvous for which there is no matching receive statement is itself not executable, and similarly for an executable receive statement with no matching executable send statement.

The process containing such a statement is blocked (unless, of course, there are alternatives with true guards in an **`if`**- or **`do`**-statement).

In the client-server program in Listing 7.1, any of the three processes can be executed first. If the client processes execute first, they will block on their send statements `request!0` (line 13) and `request!1` (line 17) until the matching receive statement in the server `request?client` (line 7) is executable.

Since both clients offer to engage in the rendezvous before the server executes the receive statement, the choice between the two send statements is made nondeterministically: randomly in random simulation mode, and in verification mode, both choices are searched.

Similarly, if the server attempts to execute the receive statement before either client offers to engage in a rendezvous, it is blocked. Since there are no other alternatives in its **do**-statement, the entire process is blocked.

Subsection 7.2.1

Reply channels

Listing 7.3. Client-server with a reply channel (`rendezvous2.pml`)

```
chan request = [0] of { byte }
chan reply = [0] of { bool }

active proctype Server() {
  byte client
end:
  do
  :: request ? client ->
       printf("Client %d\n", client)
       reply ! true
  od
}

active proctype Client0() {
  request ! 0
  reply ? _
}

active proctype Client1() {
  request ! 1
  reply ? _
}
```

Listing 7.3. Client-server with a reply channel (`rendezvous2.pml`)

```
$ spin listing_7_3.pml
      Client 1
      Client 0
      timeout
#processes: 1
  6:   proc  0 (Server:1) listing_7_3.pml:7 (state 4) <valid end
state>
3 processes created
```

Listing 7.4. Multiple clients and servers (`rendezvous3.pml`)

```
chan request = [0] of { byte }
chan reply = [0] of { byte }

active [2] proctype Server() {
  byte client
end:
  do
  :: request ? client ->
       printf("Client %d processed by server %d\n",
              client, _pid)
       reply ! _pid
  od
}

active [2] proctype Client() {
  byte server
  request ! _pid
  reply ? server
  printf("Reply received from server %d by client %d\n",
         server, _pid)
}
```

Listing 7.4. Multiple clients and servers (`rendezvous3.pml`)

```
$ spin listing_7_4.pml
          Client 2 processed by server 1
              Reply received from server 1 by client 2
      Client 3 processed by server 0
                  Reply received from server 0 by client 3
      timeout
#processes: 2
  8:   proc  1 (Server:1) listing_7_4.pml:7 (state 4) <valid end state>
  8:   proc  0 (Server:1) listing_7_4.pml:7 (state 4) <valid end state>
4 processes created

$ spin listing_7_4.pml
      Client 3 processed by server 0
      Client 2 processed by server 0
                  Reply received from server 0 by client 3
              Reply received from server 0 by client 2
      timeout
#processes: 2
  8:   proc  1 (Server:1) listing_7_4.pml:7 (state 4) <valid end state>
  8:   proc  0 (Server:1) listing_7_4.pml:7 (state 4) <valid end state>
4 processes created
```

Listing 7.4. Multiple clients and servers (`rendezvous3.pml`)

```
$ spin listing_7_4.pml
      Client 3 processed by server 0
          Client 2 processed by server 1
                  Reply received from server 1 by client 3
              Reply received from server 0 by client 2
      timeout
#processes: 2
  8:    proc  1 (Server:1) listing_7_4.pml:7 (state 4) <valid end state>
  8:    proc  0 (Server:1) listing_7_4.pml:7 (state 4) <valid end state>
4 processes created
```

Unfortunately, this program is not correct.

The model is faithful to the concept that the two servers should be independent of each other, as should the two clients. However, we need to ensure that the client receiving the reply sent by the server in line 11 is the same as the client who sent the request that was received by the server in line 8. A few random simulations of the program gave the following output that shows that each client received the reply intended fot the other:

```
Client 2 processed by server 1
Reply received from server 1 by client 3
Client 3 processed by server 0
Reply received from server 0 by client 2
```

The error can be also be found by attempting to verify the program.

Modify line 11 so that the server returns to the client the ID that it received:

```
reply ! _pid, client
```

Declare an additional local variable `whichClient` in each client to receive the second message field sent by the server (line 18):

```
reply ? server, whichClient
```

Now, add after line 20 an assertion that checks that the ID received from the server is same as the **`_pid`** of the client that sent the request:

```
assert (whichClient == _pid);
```

Spin quickly locates a computation that violates the assertion.

Listing 7.4.1. Multiple clients and servers

```
$ cat -n listing_7_4_1.pml | expand
     1  chan request = [0] of { byte }
     2  chan reply = [0] of { byte }
     3
     4  active [1] proctype Server() {
     5          byte client
     6  end:
     7          do
     8          :: request ? client ->
     9                  printf("Client %d processed by server %d\n",
    10                          client, _pid)
    11                  reply ! _pid, client
    12          od
    13  }
    14
    15  active [1] proctype Client() {
    16          byte server, whichClient
    17          request ! _pid
    18          reply ? server, whichClient
    19          printf("Reply received from server %d by client %d\n",
    20                  server, _pid)
    21          assert (whichClient == _pid)
    22  }
```

Listing 7.4.1. Multiple clients and servers

```
$ spin listing_7_4_1.pml | expand
      Client 1 processed by server 0
          Reply received from server 0 by client 1
spin: listing_7_4_1.pml:21, Error: assertion violated         <--- WHY?
spin: text of failed assertion: assert((whichClient==_pid))
#processes: 2
  5:    proc  1 (Client:1) listing_7_4_1.pml:21 (state 4)
  5:    proc  0 (Server:1) listing_7_4_1.pml:7 (state 4) <valid end state>
2 processes created
```

Listing 7.4.2. Multiple clients and servers

```
$ cat -n listing_7_4_2.pml | expand
     1  chan request = [0] of { byte }
     2  chan reply = [0] of { byte, byte }
     3
     4  active [2] proctype Server() {
     5      byte client
     6  end:
     7      do
     8      :: request ? client ->
     9          printf("Client %d processed by server %d\n",
    10                  client, _pid)
    11          reply ! _pid, client
    12      od
    13  }
    14
    15  active [2] proctype Client() {
    16      byte server, whichClient
    17      request ! _pid
    18      reply ? server, whichClient
    19      printf("Reply received from server %d by client %d\n",
    20              server, _pid)
    21      assert (whichClient == _pid)
    22  }
```

Listing 7.4.2. Multiple clients and servers

```
$ spin listing_7_4_2.pml | expand
      Client 3 processed by server 0
          Client 2 processed by server 1
                  Reply received from server 0 by client 3
              Reply received from server 1 by client 2
      timeout
#processes: 2
 10:    proc  1 (Server:1) listing_7_4_2.pml:7 (state 4) <valid end state>
 10:    proc  0 (Server:1) listing_7_4_2.pml:7 (state 4) <valid end state>
4 processes created


$ spin listing_7_4_2.pml | expand
          Client 2 processed by server 1
              Reply received from server 1 by client 2
          Client 3 processed by server 1
                  Reply received from server 1 by client 3
      timeout
#processes: 2
 10:    proc  1 (Server:1) listing_7_4_2.pml:7 (state 4) <valid end state>
 10:    proc  0 (Server:1) listing_7_4_2.pml:7 (state 4) <valid end state>
4 processes created
```

Listing 7.4.2. Multiple clients and servers

```
$ spin -a listing_7_4_2.pml
$ gcc -o pan pan.c
$ ./pan | expand
hint: this search is more efficient if pan.c is compiled -DSAFETY
pan:1: assertion violated (whichClient==_pid) (at depth 8)
pan: wrote listing_7_4_2.pml.trail

(Spin Version 6.4.3 -- 16 December 2014)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 60 byte, depth reached 14, errors: 1
       12 states, stored
        0 states, matched
       12 transitions (= stored+matched)
        0 atomic steps
...
```

Listing 7.4.2. Multiple clients and servers

```
$ spin -t -p -g -l -r -s listing_7_4_2.pml | expand
using statement merging
  1:    proc  3 (Client:1) listing_7_4_2.pml:17 Sent 3  -> queue 1 (request)
  1:    proc  3 (Client:1) listing_7_4_2.pml:17 (state 1)       [request!_pid]
  2:    proc  1 (Server:1) listing_7_4_2.pml:8 Recv 3   <- queue 1 (request)
  2:    proc  1 (Server:1) listing_7_4_2.pml:8 (state 1)        [request?client]
                Server(1):client = 3
          Client 3 processed by server 1
  3:    proc  1 (Server:1) listing_7_4_2.pml:9 (state 2)        [printf('Client %d processed
by server %d\\n',client,_pid)]
  4:    proc  2 (Client:1) listing_7_4_2.pml:17 Sent 2  -> queue 1 (request)
  4:    proc  2 (Client:1) listing_7_4_2.pml:17 (state 1)       [request!_pid]
  5:    proc  0 (Server:1) listing_7_4_2.pml:8 Recv 2   <- queue 1 (request)
  5:    proc  0 (Server:1) listing_7_4_2.pml:8 (state 1)        [request?client]
                Server(0):client = 2
      Client 2 processed by server 0
  6:    proc  0 (Server:1) listing_7_4_2.pml:9 (state 2)        [printf('Client %d processed
by server %d\\n',client,_pid)]
  7:    proc  1 (Server:1) listing_7_4_2.pml:11 Sent 1,3        -> queue 2 (reply)
  7:    proc  1 (Server:1) listing_7_4_2.pml:11 (state 3)       [reply!_pid,client]
  8:    proc  2 (Client:1) listing_7_4_2.pml:18 Recv 1,3        <- queue 2 (reply)
  8:    proc  2 (Client:1) listing_7_4_2.pml:18 (state 2)       [reply?server,whichClient]
                Client(2):whichClient = 3
                Client(2):server = 1
              Reply received from server 1 by client 2
  9:    proc  2 (Client:1) listing_7_4_2.pml:19 (state 3)       [printf('Reply received from
server %d by client %d\\n',server,_pid)]
spin: listing_7_4_2.pml:21, Error: assertion violated
spin: text of failed assertion: assert((whichClient==_pid))
  9:    proc  2 (Client:1) listing_7_4_2.pml:21 (state 4)       [assert((whichClient==_pid))]
...
```

Listing 7.4.2. Multiple clients and servers

```
...
spin: trail ends after 9 steps
#processes: 4
  9:    proc  3 (Client:1) listing_7_4_2.pml:18 (state 2)
  9:    proc  2 (Client:1) listing_7_4_2.pml:22 (state 5) <valid end state>
  9:    proc  1 (Server:1) listing_7_4_2.pml:7 (state 4) <valid end state>
  9:    proc  0 (Server:1) listing_7_4_2.pml:11 (state 3)
4 processes created
```

Subsection 7.2.2

Arrays of channels

One way to fix the above error is to associate a separate reply channel with each client. Listing 7.5 is similar to Listing 7.4 with several changes:

- The `reply` channel is changed to be an array of two channels (line 2), one for each client.
- The messages on the `request` channels include a field of type **`chan`** for the reply channel in addition to the field of type **`byte`** for the client ID (line 1).
- A client sends the reply channel associated with it (in addition to its ID) (line 19) and a server receives the value and stores it in the variable `replyChannel` (line 20).

- The server uses the received value of the reply channel to ensure that the reply is sent to the correct client (line 12).

- The client waits for a reply on the channel associated with it (line 20).

Running a verification in Spin shows that the assertion is never violated.

The use of **`_pid`**`-2` to obtain the index of the channel in the client processes (lines 19-20) is not a robust programming technique, because the **`_pid`** will change if an additional process is declared before the client processes. The software archive contains a version of this program (`rendezvous4-verify.pml`) where the IDs and indices are passed as parameters of a **`run`** operator, and thus are less likely to need modification.

Listing 7.5. An array of channels (`rendezvous4.pml`)

```
chan request = [0] of { byte, chan }
chan reply [2] = [0] of { byte, byte }

active [2] proctype Server() {
  byte client
  chan replyChannel
end:
  do
  :: request ? client, replyChannel ->
       printf("Client %d processed by server %d\n",
              client, _pid)
       replyChannel ! _pid, client
  od
}

active [2] proctype Client() {
  byte server
  byte whichClient
  request ! _pid, reply[_pid-2]
  reply[_pid-2] ? server, whichClient
  printf("Reply received from server %d by client %d\n",
         server, _pid)
  assert(whichClient == _pid)
}
```

Listing 7.5. An array of channels (`rendezvous4.pml`)

```
$ spin -a rendezvous4.pml
$ gcc -o pan pan.c
$ ./pan | expand
hint: this search is more efficient if pan.c is compiled -DSAFETY

(Spin Version 6.4.3 -- 16 December 2014)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 68 byte, depth reached 14, errors: 0
       71 states, stored
        8 states, matched
       79 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.007       equivalent memory usage for states (stored*(State-vector + overhead))
    0.292       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage
...
```

Subsection 7.2.3

Local channels

Local channels (`listing_7_2_3.pml`)

```
$ cat -n listing_7_2_3.pml | expand
     1  chan request = [0] of { byte, chan }
     2
     3  active [2] proctype Server() {
     4      byte client
     5      chan replyChannel
     6  end:
     7      do
     8      :: request ? client, replyChannel ->
     9          printf("Client %d processed by server %d\n",
    10                  client, _pid)
    11          replyChannel ! _pid, client
    12      od
    13  }
    14
    15  active [2] proctype Client() {
    16      byte server, whichClient
    17      chan reply = [0] of { byte, byte }
    18      request ! _pid, reply
    19      reply ? server, whichClient
    20      printf("Reply received from server %d by client %d\n",
    21              server, _pid)
    22      assert (whichClient == _pid)
    23  }
```

Local channels (`listing_7_2_3.pml`)

```
$ spin -a listing_7_2_3.pml
$ gcc -o pan pan.c
$ ./pan | expand
hint: this search is more efficient if pan.c is compiled -DSAFETY

(Spin Version 6.4.3 -- 16 December 2014)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 68 byte, depth reached 14, errors: 0
       71 states, stored
        8 states, matched
       79 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

...
```

Subsection 7.2.4

Limitations of rendezvous channels

... If rendezvous channels were used, the number of clients actually being served can be no larger than the number of servers, so the rest of the clients would be blocked.

We can show this by counting the number of clients in our program that have successfully sent the request but have yet to receive the reply. With two servers and four clients, there can be at most two clients in this state.

(rendezvous4-verify.pml)

```
$ cat -n rendezvous4-verify.pml | expand
   ...
     3  chan request = [4] of { byte, chan }
     4  chan reply [4] = [0] of { byte }
     5  byte numClients = 0
     6
     7  active [2] proctype Server() {
     8      byte client
     9      byte me = _pid
    10      chan replyChannel
    11  end:
    12      do
    13      :: request ? client, replyChannel ->
    14          printf("Client %d processed by server %d\n", client, me)
    15          replyChannel ! me
    16      od
    17  }
    18
    19  active [4] proctype Client() {
    20      byte server
    21      byte me = _pid - 2
    22      request ! me, reply[me]
    23      numClients++
    24      assert (numClients <= 2)
    25      numClients--
    26      reply[me] ? server
    27      printf("Reply received from server %d by client %d\n", server, me)
    28  }
```

(rendezvous4-verify.pml)

```
$ spin -a rendezvous4-verify.pml
$ gcc -o pan pan.c
$ ./pan | expand
hint: this search is more efficient if pan.c is compiled -DSAFETY
pan:1: assertion violated (numClients<=2) (at depth 12)         <--- WHY?
pan: wrote rendezvous4-verify.pml.trail

(Spin Version 6.4.3 -- 16 December 2014)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 108 byte, depth reached 40, errors: 1
     4165 states, stored
     2816 states, matched
     6981 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

...
```