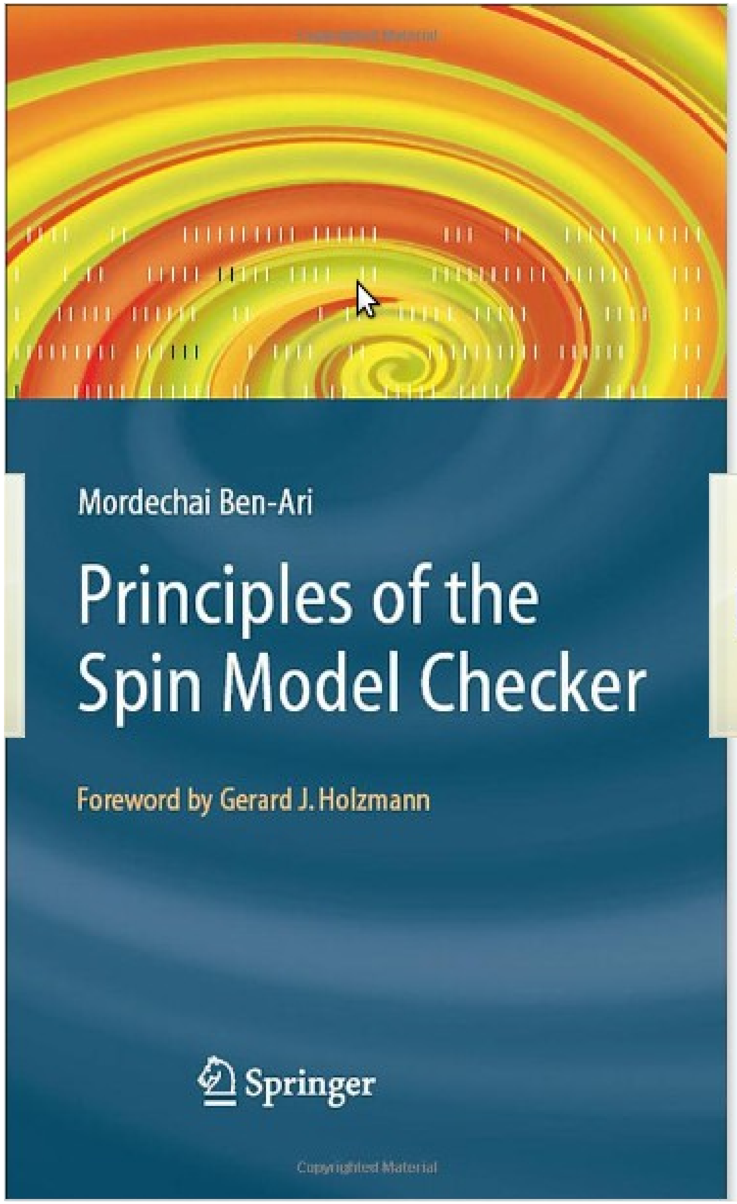
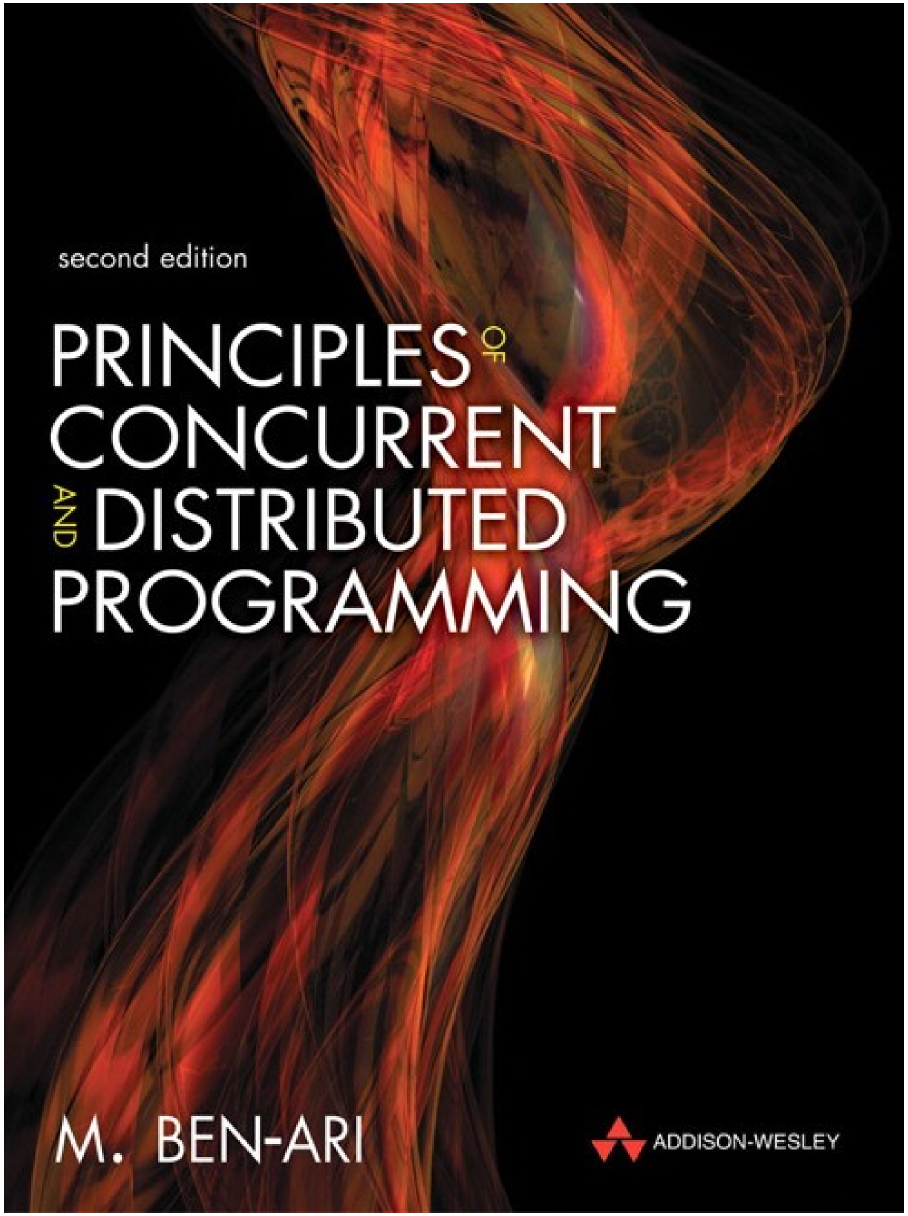

Copyrighted Material
Mordechai Ben-Ari
Principles of the Spin Model Checker
Foreword by Gerard J. Holzmann
Springer
Copyrighted Material

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

Section 7.3

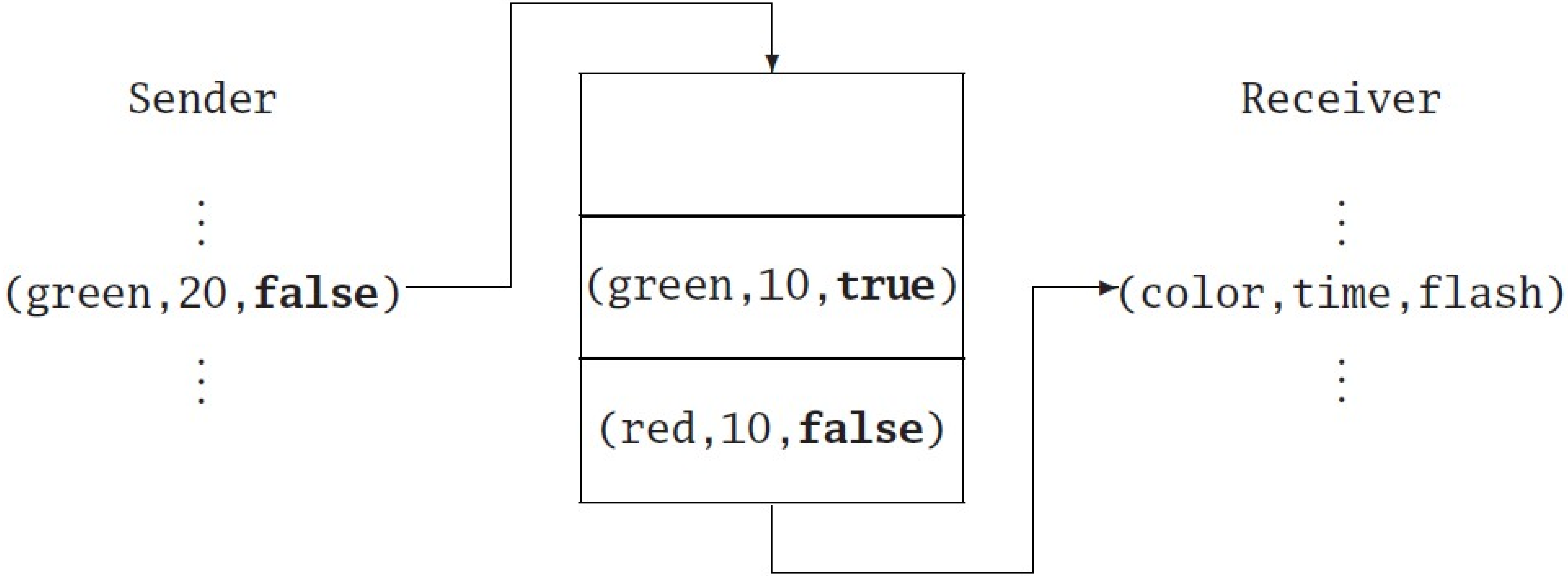
Buffered channels

A channel declared with a positive capacity is called a *buffered channel*:

```
    chan ch = [3] of { mtype, byte, bool };
```

The capacity is the number of messages of the message type that can be stored in the channel.

The send and receive statements treat the channel as a FIFO (first in-first out) queue. (Other versions of the send and receive statements are described Sections 7.5-7.8.)



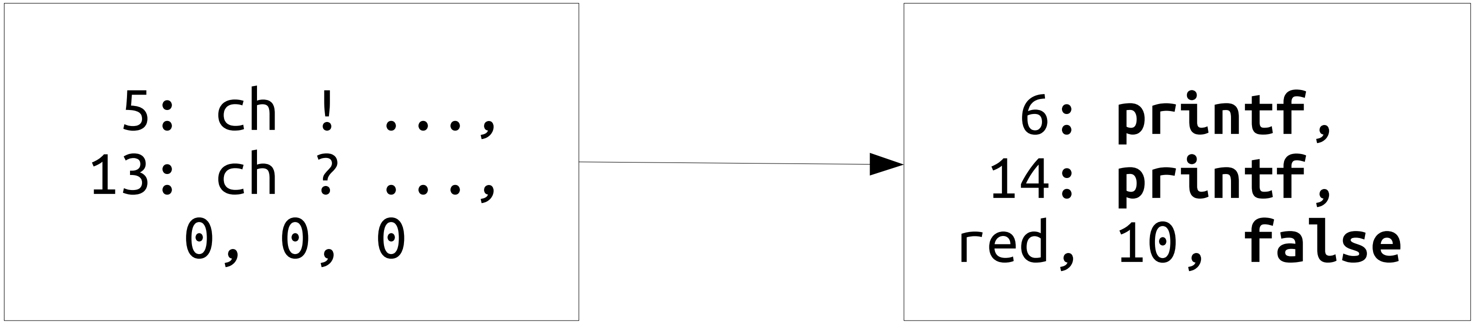
Sender
⋮
(green,20,**false**)
⋮
(green,10,**true**)
(red,10,**false**)
Receiver
⋮
(color,time,flash)
⋮

The diagram shows the channel as it would appear if two messages have already been sent to the channel; more precisely, it shows the channel after two *more* messages have been sent than have been received.

The send statement is executable because there is room in the channel for another message, that is, the channel is *not full*; executing the statement places the message at the tail of the queue.

The receive statement is executable because there are messages in the channel, that is, the channel is *not empty*; executing the statement removes the message at the head of the queue and assigns its values to the variables in the receive command.

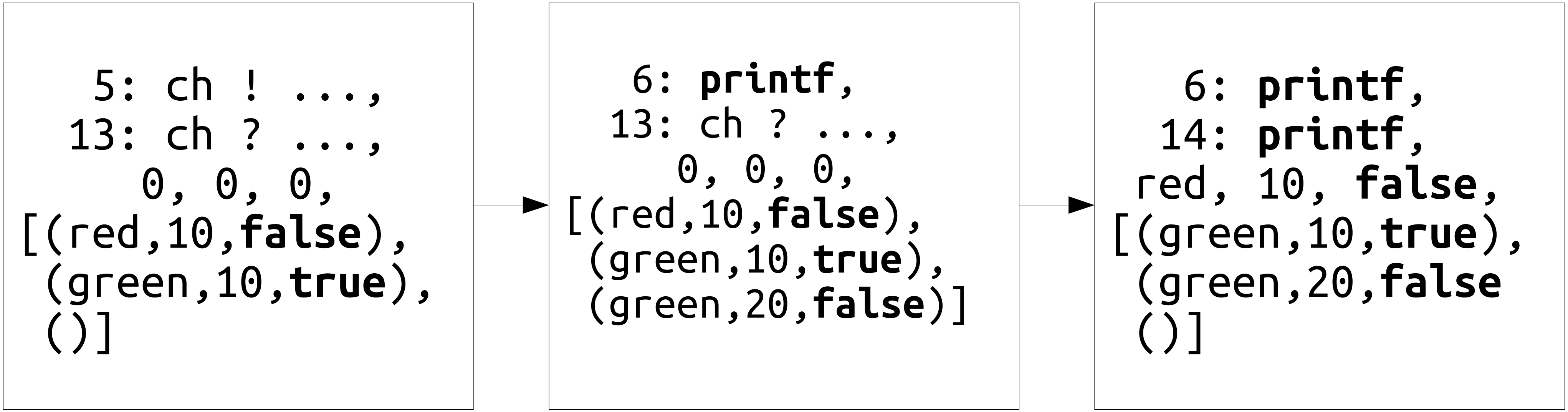
The channel is part of the states of the computation. The send and receive statements are each executed atomically. The state diagram corresponding to



```
 5: ch ! ...,
13: ch ? ...,
  0, 0, 0
```

```
 6: printf,
14: printf,
red, 10, false
```

is:



```
  5: ch ! ...,
 13: ch ? ...,
   0, 0, 0,
[(red,10,false),
 (green,10,true),
 ()]
```

```
  6: printf,
 13: ch ? ...,
    0, 0, 0,
[(red,10,false),
 (green,10,true),
 (green,20,false)]
```

```
  6: printf,
 14: printf,
red, 10, false,
[(green,10,true),
 (green,20,false
 ()]
```

The channel is shown as a triple `[_,_,_]` with spaces for three elements.

Advanced: Lost messages when the channel is full

The semantics of the send statement can be changed by running Spin with `-m` argument:

```
$ spin -m ...

(-m   lose msgs sent to full queues)
```

Instead of blocking when the channel is full, the send statement is executed but the content of the channel does not change so that the message is lost.

Section 7.4

Checking the content of a channel

Warning

The functions described in this section are allowed only for buffered channels, because it makes no sense to talk about "the number of messages" in a rendezvous channel.

Subsection 7.4.1

Checking if a channel is full or empty

There are four predefined boolean functions for checking a channel:

full and
empty,

and their negations

nfull and
nempty.

```
active proctype Foo() {
    chan q = [8] of { byte }
    byte one_more = 0

    assert(len(q) == 0)
    assert(empty(q))
    assert(nfull(q))
    do
    :: q!one_more; one_more++   /* send messages */
    :: full(q) -> break         /* untill full   */
    od
    assert(len(q) == 8)
    assert(full(q))
    assert(nempty(q))
}
```

Listing 7.6. Checking if the channel is full or empty (`channel3.pml`)

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

active [2] proctype Server() {
  byte client
  chan replyChannel
  do
  :: empty(request) ->
       printf("No requests for server %d\n", _pid)
  :: request ? client, replyChannel ->
       printf("Client %d processed by server %d\n", client, _pid)
       replyChannel ! _pid
  od
}

active [2] proctype Client() {
  byte server
  do
  :: nfull(request) ->
       printf("Client %d waiting for non-full channel\n", _pid)
  :: request ! _pid, reply[_pid-2] ->
       reply[_pid-2] ? server
       printf("Reply received from server %d by client %d\n", server, _pid)
  od
}
```

Listing 7.6. Checking if the channel is full or empty (`channel3.pml`)

```
$ spin -u30 channel3.pml | expand
      No requests for server 0
                  Client 3 waiting for non-full channel
          No requests for server 1
              Client 2 waiting for non-full channel
      No requests for server 0
          Client 3 processed by server 1
                  Reply received from server 1 by client 3
      Client 2 processed by server 0
          No requests for server 1
-------------
depth-limit (-u30 steps) reached
#processes: 4
                queue 1 (request):
                queue 2 (reply[0]): [0]
                queue 3 (reply[1]):
 30:    proc  3 (Client:1) channel3.pml:22 (state 2)
 30:    proc  2 (Client:1) channel3.pml:24 (state 4)
 30:    proc  1 (Server:1) channel3.pml:16 (state 7)
 30:    proc  0 (Server:1) channel3.pml:16 (state 7)
4 processes created
```

Warning

Do not use **`else`** alternatives in **`if`**- or **`do`**-statements that have channel expressions as their guards; instead, use the pair **`empty`**/**`nempty`** and **`full`**/**`nfull`**.

Subsection 7.4.2

Checking the number of messages in a channel

There is a predefined integer function `len(ch)` that returns the number of messages in channel `ch`.

This is less usefull than it may seem at first glance because most models need only check the extreme cases of `len(ch)==0` and `len(ch)==N` (where `N` is the capacity of `ch`), for which the functions of the previous subsection suffice.

One example of its use is in a model of process allocation: when the channel is more than three-quarters full, allocate more server processes to service the overload, and when the channel is less than one-quarter full, some processors can be deallocated:

```
if
:: len(ch) > (3*N/4) -> /* Allocate a new server */
:: len(ch) < (N/4)   -> /* Deallocate a server */
:: else
fi
```

Section 7.5

Random receive

Buffered channels can be used to implement a different solution to the client-server problem (Listing 7.7, next slide), in which the array of four reply channels has been replaced by a single channel of capacity four (line 4). The message sent on the reply channel (line 12) contains the server ID, as well as the ID of the client that was received from the request channel (line 10). We need to ensure that it is possible for a client to receive only messages meant for it, but this cannot be done with the channel statements as we have defined them so far.

The first problem is that channels are FIFO, so even if a channel contains a message for a certain client, the client cannot receive the message until all messages ahead of it in the queue have been removed from the channel. To solve this problem we can use the Promela statement called *random receive*, which receives messages from *anywere* within the channel, not just at the head; it is denoted by the double question mark as shown in the line 19.

Listing 7.7. Random receive from a buffered channel (channel2.pml)

```
chan request = [4] of { byte }
chan reply = [4] of { byte, byte }

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

active [4] proctype Client() {
  byte server
  request ! _pid
  reply ?? server, eval(_pid)
  printf("Reply received from server %d by client %d\n", server, _pid)
}
```

The second problem is that the receive statement removes a message regardless of its content and assigns the values to the variables in the statement. Here it is required that a client remove only messages intended for itself.

To solve this problem the receive statement allows *values* to be used instead of variables.

A receive statement is executable if and only if its variables and values *match* the values in the fields of a message. A variable matches any field whose value is of the correct type, but a value matches a field if and only if it equals the value of the field.

When the message is received, it is removed from the channel and values are assigned to the variables in the statement; of course, there is no meaning to assigning values to values.

A random receive statement will remove the *first* message that matches the variables and values in the statement. If the value to be matched is known when the program is written (for a example, the client ID is a constant), it can be used directly:

```
    reply ?? server, 3
```

But sometimes the value is known only at runtime, for example, when it is the value of a parameter to the **proctype**, or, as in this case, when it is the value of **_pid** that is different for each instantiation of the **proctype**. In these cases, **eval** is used to obtain *the current value of the variable* to use in the matching (line 19):

```
    reply ?? server, eval(_pid)
```

This ensures that only messages intended for this client are matched and are removed from the channel.

NAME

eval - predefined unary function to turn an expression into a constant.

SYNTAX

eval(*any_expr*)

DESCRIPTION

The intended use of eval is in receive statements to force a match of a message field with the current value of a local or global variable. Normally, such a match can only be forced by specifying a constant. If a variable name is used directly, without the eval function, the variable would be assigned the value from the corresponding message field, instead of serving as a match of values.

EXAMPLES

In the following example the two receive operations are only executable if the precise values specified were sent to channel q : first an ack and then a msg .

```
    mtype = { msg, ack, other };
    chan q = [4] of { mtype };

    mtype x;

    x = ack; q?eval(x)/* same as: q?ack */
    x = msg; q?eval(x)/* same as: q?msg */
```

Without the eval function, writing simply

```
    q?x
```

would mean that whatever value was sent to the channel (e.g., the value other) would be assigned to x when the receive operation is executed.

...

...

NOTES

Any expression can be used as an argument to the eval function. The result of the evaluation of the expression is then used as if it were a constant value.

This mechanism can also be used to specify a conditional rendezvous operation, for instance by using the value true in the sender and using a conditional expression with an eval function at the receiver; see also the manual page for conditional expressions.

Listing 7.7. Random receive from a buffered channel (`channel2.pml`)

```
$ spin channel2.pml | expand
          Client 3 processed by server 1
      Client 5 processed by server 0
                  Reply received from server 1 by client 3
          Client 2 processed by server 1
              Reply received from server 1 by client 2
          Client 4 processed by server 1
                      Reply received from server 1 by client 4
                          Reply received from server 0 by client 5
      timeout
#processes: 2
                queue 1 (request):
                queue 2 (reply):
 28:    proc  1 (Server:1) channel2.pml:9 (state 4) <valid end state>
 28:    proc  0 (Server:1) channel2.pml:9 (state 4) <valid end state>
6 processes created
```

Section 7.6

Sorted send

A send statement for a buffered channel inserts the message at the tail of the message queue in the channel. With the *sorted send* statement, written `ch!!message` with a double exclamation point, the message is inserted *ahead* of the first message that is larger than it.

Fields of the message are interpreted as integer values, and if there are multiple fields, lexicographic ordering is used.

Sorted send can be used to model a data structure such as a priority queue. The program in Listing 7.8 prints three values in sorted order, even though they are generated nondeterministically. For another example of the use of sorted send, see Section 11.3.

Listing 7.8. Storing values in sorted order (sorted.pml)

```
chan ch = [3] of { byte }

inline getValue() {
  if
  :: n = 1
  :: n = 2
  :: n = 3
  fi
}

active proctype Sort() {
  byte n
  getValue(); ch !! n
  getValue(); ch !! n
  getValue(); ch !! n
  ch ? n; printf("%d\n", n)
  ch ? n; printf("%d\n", n)
  ch ? n; printf("%d\n", n)
}
```

Listing 7.8. Storing values in sorted order (`sorted.pml`)

```
$ spin sorted.pml | expand
      1
      2
      3
1 process created

$ spin sorted.pml | expand
      1
      3
      3
1 process created

$ spin sorted.pml | expand
      2
      2
      3
1 process created
```

Section 7.7

Copying the value of a message

Sometimes we are interested in copying the values in a message without removing the message from the channel. The following statements copy the values of a message into the three variables but does not remove it:

```
      ch ? <color, time, flash>
      ch ?? <color, time, flash>
```

This statement is distinguished from normal and random receive statements by the use of angle brackets to enclose a list of variables.

Copying without removing is useful when channels are used to implement data structures, as described in Section 11.1.

Section 7.8

Polling

Real-time systems are built in two architectural styles:

- In an interrupt-driven system, a sensor causes an interrupt of the CPU whenever data are ready to be read.
- In a polling system, sensors are periodically checked by the CPU to see if data are ready to be read.

Interrupt-driven systems are modeled with blocking receive statements. To model polling systems, Promela supports polling receive statements. Only buffered channels can be polled.

Polling receive statements are not the same as receive statements that do not remove messages from a channel:

```
    ch ?? <green, time, false>
```

Although the message is not removed from the channel, copying values into the variables creates a side-effect so the statement cannot be used in a guard.

A polling expression (written with square brackets) is *side-effect free* and can be used in a guard:

```
do
:: ch ?? [green, time, false] ->
     ch ?? green, time, false
:: else -> /* Do something else */
od
```

It can also be used in a subexpression of a compound expression, as shown in the following code where we check the channel only on even-numbered executions of the loop body:

```
bool even = true

do
:: even && ch ?? [green, time, false] ->
    ch ?? green, time, false;
    even = !even
:: else ->
    /* Do something else */
    even = !even
od
```

Since the evaluation of a guard and the execution of the first statement after the guard are two separate atomic operations, if other processes also receive from the channel, it is possible that the poll statement

```
      ch ?? [green, time, false]
```

will be true, but that due to interleaving, the receive statement

```
      ch ?? green, time, false
```

will no longer be executable. This can be solved by placing the **do**-statement within **atomic**.

Section 7.9

Comparing rendezvous and buffered channels

The choice between using rendezvous or buffered channels in a model depends on several factors, so the analysis of the tradeoffs between them is a significant aspect of modeling a system.

Rendezvous channels are far more efficient. There is no “variable” associated with a rendezvous channel, so using one does not increase the size of a state. Buffered channels, on the other hand, greatly increase the potential size of the state space because every permutation of messages up to the capacity of the channel might occur in a computation, and the messages themselves can have multiple fields. Furthermore, rendezvous channels are unique in that a single step of a computation causes changes in values of the location counters of more than one process.

In a sense, a buffered channel is just a convenience because it could be implemented with rendezvous channels and an additional process to store the contents of the channel. Programming languages like Occam and Ada take precisely this approach and support only communication by rendezvous.

However, the convenience of using buffered channels contributes significantly to the ease of constructing models. They facilitate modeling communications systems that contain channels that can store messages. Buffered channels also enable direct modeling of asynchronous systems where processes transfer data without blocking.

When a buffered channel is used in a model, the channel capacity must be carefully considered. A large capacity may be more realistic, but can cause an explosion in the size of the state space that can make verification impractical.
Section 11.8 gives an example of how to choose the channel capacity to enable verification.