

Formal Methods for Software Development

Modeling Distributed Systems

Wolfgang Ahrendt

08 September 2017

This Lecture

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done. –Leslie Lamport

Using PROMELA channels for modeling distributed systems

Modeling Distributed Systems

Distributed systems consist of

- ▶ nodes,
- ▶ interacting via communication channels,
- ▶ protocols dictate how nodes communicate with each other.

Modeling Distributed Systems

Distributed systems consist of

- ▶ **nodes**,
- ▶ interacting via **communication channels**,
- ▶ **protocols** dictate how nodes communicate with each other.

Distributed systems are very complex.

Models of distributed systems abstract away from details of networks/protocols/nodes.

In PROMELA:

- ▶ **nodes** modeled by **PROMELA processes**
- ▶ **communication channels** modeled by **PROMELA channels**
- ▶ protocols modeled by algorithm distributed over processes

Channels in PROMELA

In PROMELA, channels are first class citizens.

Data type `chan` with two operations for **sending** and **receiving**

Channels in PROMELA

In PROMELA, channels are first class citizens.

Data type `chan` with two operations for **sending** and **receiving**

A variable of channel type is declared by initializer:

```
chan name = [capacity] of {type1, ..., typen}
```

<i>name</i>	name of channel variable
<i>capacity</i>	non-negative integer constant
<i>type</i> _{<i>i</i>}	PROMELA data types

Channels in PROMELA

In PROMELA, channels are first class citizens.

Data type `chan` with two operations for **sending** and **receiving**

A variable of channel type is declared by initializer:

```
chan name = [capacity] of {type1, ..., typen}
```

name name of channel variable

capacity non-negative integer constant

*type*_{*i*} PROMELA data types

Example:

```
chan ch = [2] of { mtype, byte, bool }
```

Meaning of Channels

chan *name* = [*capacity*] of {*type*₁, ..., *type*_{*n*}}

Creates channel, stored in variable *name*

Meaning of Channels

`chan name = [capacity] of {type1, ..., typen}`

Creates channel, stored in variable *name*

Messages communicated via channel are *n*-tuples $\in \textit{type}_1 \times \dots \times \textit{type}_n$

Meaning of Channels

`chan name = [capacity] of {type1, ..., typen}`

Creates channel, stored in variable *name*

Messages communicated via channel are *n*-tuples $\in \text{type}_1 \times \dots \times \text{type}_n$

Can buffer up to *capacity* messages, if *capacity* ≥ 1

\Rightarrow “buffered channel”

Meaning of Channels

`chan name = [capacity] of {type1, ..., typen}`

Creates channel, stored in variable *name*

Messages communicated via channel are *n*-tuples $\in \text{type}_1 \times \dots \times \text{type}_n$

Can buffer up to *capacity* messages, if *capacity* ≥ 1

\Rightarrow "buffered channel"

The channel has *no* buffer if *capacity* = 0

\Rightarrow "rendezvous channel"

Meaning of Channels

Example:

```
chan ch = [2] of { mtype, byte, bool }
```

Creates channel, stored in variable ch

Meaning of Channels

Example:

```
chan ch = [2] of { mtype, byte, bool }
```

Creates channel, stored in variable `ch`

Messages communicated via `ch` are 3-tuples $\in \mathbf{mtype} \times \mathbf{byte} \times \mathbf{bool}$

Meaning of Channels

Example:

```
chan ch = [2] of { mtype, byte, bool }
```

Creates channel, stored in variable `ch`

Messages communicated via `ch` are 3-tuples $\in \text{mtype} \times \text{byte} \times \text{bool}$

Given, e.g., `mtype = {red, yellow, green}`,
an example message on `ch` can be:

Meaning of Channels

Example:

```
chan ch = [2] of { mtype, byte, bool }
```

Creates channel, stored in variable `ch`

Messages communicated via `ch` are 3-tuples $\in \text{mtype} \times \text{byte} \times \text{bool}$

Given, e.g., `mtype = {red, yellow, green}`,
an example message on `ch` can be: `green, 20, false`

Meaning of Channels

Example:

```
chan ch = [2] of { mtype, byte, bool }
```

Creates channel, stored in variable `ch`

Messages communicated via `ch` are 3-tuples $\in \text{mtype} \times \text{byte} \times \text{bool}$

Given, e.g., `mtype = {red, yellow, green}`,
an example message on `ch` can be: `green, 20, false`

`ch` is a *buffered channel*, buffering up to 2 messages

Sending and Receiving

send statement has the form:

$$name ! expr_1, \dots, expr_n$$

Sending and Receiving

send statement has the form:

$name ! expr_1, \dots, expr_n$

- ▶ $name$: channel variable

Sending and Receiving

send statement has the form:

name ! *expr*₁, ... , *expr*_{*n*}

- ▶ *name*: channel variable
- ▶ *expr*₁, ... , *expr*_{*n*}: sequence of expressions, where number and types match message type

Sending and Receiving

send statement has the form:

name ! expr₁, ... , expr_n

- ▶ *name*: channel variable
- ▶ *expr₁, ... , expr_n*: sequence of expressions, where number and types match message type
- ▶ sends *values* of *expr₁, ... , expr_n* as *one* message

Sending and Receiving

send statement has the form:

name ! expr₁, ... , expr_n

- ▶ *name*: channel variable
- ▶ *expr₁, ... , expr_n*: sequence of expressions, where number and types match message type
- ▶ sends *values* of *expr₁, ... , expr_n* as *one* message
- ▶ example: `ch ! green, i+20, false`

Sending and Receiving

send statement has the form:

name ! expr₁, ... , expr_n

- ▶ *name*: channel variable
- ▶ *expr₁, ... , expr_n*: sequence of expressions, where number and types match message type
- ▶ sends *values* of *expr₁, ... , expr_n* as *one* message
- ▶ example: `ch ! green, i+20, false`

receive statement has the form:

name ? var₁, ... , var_n

Sending and Receiving

send statement has the form:

name ! expr₁, ... , expr_n

- ▶ *name*: channel variable
- ▶ *expr₁, ... , expr_n*: sequence of expressions, where number and types match message type
- ▶ sends *values* of *expr₁, ... , expr_n* as *one* message
- ▶ example: `ch ! green, i+20, false`

receive statement has the form:

name ? var₁, ... , var_n

- ▶ *name*: channel variable

Sending and Receiving

send statement has the form:

name ! *expr*₁, ... , *expr*_{*n*}

- ▶ *name*: channel variable
- ▶ *expr*₁, ... , *expr*_{*n*}: sequence of expressions, where number and types match message type
- ▶ sends *values* of *expr*₁, ... , *expr*_{*n*} as *one* message
- ▶ example: `ch ! green, i+20, false`

receive statement has the form:

name ? *var*₁, ... , *var*_{*n*}

- ▶ *name*: channel variable
- ▶ *var*₁, ... , *var*_{*n*}: sequence of variables, where number and types match message type

Sending and Receiving

send statement has the form:

name ! *expr*₁, ... , *expr*_{*n*}

- ▶ *name*: channel variable
- ▶ *expr*₁, ... , *expr*_{*n*}: sequence of expressions, where number and types match message type
- ▶ sends *values* of *expr*₁, ... , *expr*_{*n*} as *one* message
- ▶ example: `ch ! green, i+20, false`

receive statement has the form:

name ? *var*₁, ... , *var*_{*n*}

- ▶ *name*: channel variable
- ▶ *var*₁, ... , *var*_{*n*}: sequence of variables, where number and types match message type
- ▶ *assigns* values of message to *var*₁, ... , *var*_{*n*}

Sending and Receiving

send statement has the form:

name ! expr₁, ... , expr_n

- ▶ *name*: channel variable
- ▶ *expr₁, ... , expr_n*: sequence of expressions, where number and types match message type
- ▶ sends *values* of *expr₁, ... , expr_n* as *one* message
- ▶ example: `ch ! green, i+20, false`

receive statement has the form:

name ? var₁, ... , var_n

- ▶ *name*: channel variable
- ▶ *var₁, ... , var_n*: sequence of variables, where number and types match message type
- ▶ *assigns* values of message to *var₁, ... , var_n*
- ▶ example: `ch ? color, time, flash`

Client-Server

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

```
active proctype Client0() {  
  request ! 0  
}
```

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

```
...
```

Client-Server

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

```
active proctype Client0() {  
    request ! 0  
}
```

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

...

Client0 and Client1 send messages 0 resp. 1 to channel request

Client-Server

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

```
active proctype Client0() {  
  request ! 0  
}
```

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

...

Client0 and Client1 send messages 0 resp. 1 to channel request

Order of sending is nondeterministic

Client-Server

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

```
...
```

```
active proctype Server() {  
    byte num;  
    do  
        :: request ? num;  
        printf("serving client %d\n", num)  
    od  
}
```

Client-Server

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

```
...
```

```
active proctype Server() {  
    byte num;  
    do  
        :: request ? num;  
        printf("serving client %d\n", num)  
    od  
}
```

Server loops on

Client-Server

```
chan request = [0] of { byte };  
  
...  
  
active proctype Server() {  
    byte num;  
    do  
        :: request ? num;  
        printf("serving client %d\n", num)  
    od  
}
```

Server loops on

- ▶ receiving first message from request,

Client-Server

```
chan request = [0] of { byte };  
  
...  
  
active proctype Server() {  
    byte num;  
    do  
        :: request ? num;  
        printf("serving client %d\n", num)  
    od  
}
```

Server loops on

- ▶ receiving first message from request, storing value in num

Client-Server

```
chan request = [0] of { byte };  
  
...  
  
active proctype Server() {  
    byte num;  
    do  
        :: request ? num;  
        printf("serving client %d\n", num)  
    od  
}
```

Server loops on

- ▶ receiving first message from request, storing value in num
- ▶ printing

rendezvous1
random simulation

Executability of receive Statement (non-buffered)

request ? num

executable only when another process offers send on channel request

Executability of receive Statement (non-buffered)

`request ? num`

executable only when another process offers send on channel request

⇒ receive statement frequently used as guard in `if/do`-statements

Executability of receive Statement (non-buffered)

```
request ? num
```

executable only when another process offers send on channel request

⇒ receive statement frequently used as guard in `if/do`-statements

```
do
  :: request ? num ->
    printf("serving client %d\n", num)
od
```

Executability of receive Statement (non-buffered)

```
request ? num
```

executable only when another process offers send on channel request

⇒ receive statement frequently used as guard in `if/do`-statements

```
do
  :: request ? num ->
    printf("serving client %d\n", num)
od
```

("->" equivalent to ";")

rendezvous1
interactive simulation

Rendezvous Channels

```
chan ch = [0] of { byte, byte };

/* global only to make visible in SpinSpider */
byte hour, minute;

active proctype Sender() {
    printf("ready\n");
    ch ! 11, 45;
    printf("Sent\n")
}

active proctype Receiver() {
    printf("steady\n");
    ch ? hour, minute;
    printf("Received\n")
}
```

Rendezvous Channels

```
chan ch = [0] of { byte, byte };

/* global only to make visible in SpinSpider */
byte hour, minute;

active proctype Sender() {
    printf("ready\n");
    ch ! 11, 45;
    printf("Sent\n")
}

active proctype Receiver() {
    printf("steady\n");
    ch ? hour, minute;
    printf("Received\n")
}
```

Which interleavings can occur?

Rendezvous Channels

```
chan ch = [0] of { byte, byte };

/* global only to make visible in SpinSpider */
byte hour, minute;

active proctype Sender() {
    printf("ready\n");
    ch ! 11, 45;
    printf("Sent\n")
}

active proctype Receiver() {
    printf("steady\n");
    ch ? hour, minute;
    printf("Received\n")
}
```

Which interleavings can occur? \Rightarrow ask SPINSPIDER

through JSPIN:
SPINSPIDER on ReadySteady.pml

Rendezvous are Synchronous

On a rendezvous channel:

Transfer of message from sender to receiver is **synchronous**,
i.e., **one single operation**.

Rendezvous are Synchronous

On a rendezvous channel:

Transfer of message from sender to receiver is **synchronous**,
i.e., **one single operation**.

Sender		Receiver
⋮		⋮
(11,45)	→	(hour,minute)
⋮		⋮

Rendezvous are Synchronous

Either:

1. Location counter of sender process at send ("!"): *"offer to engage in rendezvous"*

Rendezvous are Synchronous

Either:

1. Location counter of sender process at send ("!"): *"offer to engage in rendezvous"*
2. Location counter of receiver process at receive ("?"): *"rendezvous can be accepted"*

Rendezvous are Synchronous

Either:

1. Location counter of sender process at send ("!"): *"offer to engage in rendezvous"*
2. Location counter of receiver process at receive ("?"): *"rendezvous can be accepted"*

or the other way round:

1. Location counter of receiver process at receive ("?"): *"offer to engage in rendezvous"*

Rendezvous are Synchronous

Either:

1. Location counter of sender process at send ("!"): *"offer to engage in rendezvous"*
2. Location counter of receiver process at receive ("?"): *"rendezvous can be accepted"*

or the other way round:

1. Location counter of receiver process at receive ("?"): *"offer to engage in rendezvous"*
2. Location counter of sender process at send ("!"): *"rendezvous can be accepted"*

Rendezvous are Synchronous

Either:

1. Location counter of sender process at send ("!"): *"offer to engage in rendezvous"*
2. Location counter of receiver process at receive ("?"): *"rendezvous can be accepted"*

or the other way round:

1. Location counter of receiver process at receive ("?"): *"offer to engage in rendezvous"*
2. Location counter of sender process at send ("!"): *"rendezvous can be accepted"*

In both case, the next step is:

Location counter of **both** processes is incremented at once.

Rendezvous are Synchronous

Either:

1. Location counter of sender process at send ("!"): *"offer to engage in rendezvous"*
2. Location counter of receiver process at receive ("?"): *"rendezvous can be accepted"*

or the other way round:

1. Location counter of receiver process at receive ("?"): *"offer to engage in rendezvous"*
2. Location counter of sender process at send ("!"): *"rendezvous can be accepted"*

In both case, the next step is:

Location counter of **both** processes is incremented at once.

Only place where PROMELA processes execute synchronously

Reconsider Client Server

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

active proctype Server() {
    byte num;
    do :: request ? num ->
        printf("serving client %d\n", num)
    od
}

active proctype Client0() {
    request ! 0
}

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

Reconsider Client Server

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

active proctype Server() {
  byte num;
  do :: request ? num ->
    printf("serving client %d\n", num)
  od
}

active proctype Client0() {
  request ! 0
}

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

So far **no reply** to clients

Reply Channels

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

active proctype Server() {
    byte num;
    do :: request ? num ->
        printf("serving client %d\n", num);
        ack ! true
    od
}

active proctype Client0() {
    request ! 0; ack ? _; printf("acknowledged\n")
}

active proctype Client1() {
    request ! 1; ack ? _; printf("acknowledged\n")
}
```

Reply Channels

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

active proctype Server() {
    byte num;
    do :: request ? num ->
        printf("serving client %d\n", num);
        ack ! true
    od
}

active proctype Client0() {
    request ! 0; ack ? _; printf("acknowledged\n")
}

active proctype Client1() {
    request ! 1; ack ? _; printf("acknowledged\n")
}
```

(Anonymous variable “_”: data from channel no stored anywhere)

Reply Channels - Single Server

```
mtype = { nice, rude };
chan request = [0] of { mtype };
chan reply = [0] of { mtype };

active proctype Server() {
    mtype msg;
    do :: request ? msg; reply ! msg
    od
}

active proctype NiceClient() {
    mtype msg;
    request ! nice; reply ? msg;
}

active proctype RudeClient() {
    mtype msg;
    request ! rude; reply ? msg
}
```

Reply Channels - Single Server

```
mtype = { nice, rude };
chan request = [0] of { mtype };
chan reply = [0] of { mtype };

active proctype Server() {
    mtype msg;
    do :: request ? msg; reply ! msg
    od
}

active proctype NiceClient() {
    mtype msg;
    request ! nice; reply ? msg;
    assert(msg == nice)
}

active proctype RudeClient() {
    mtype msg;
    request ! rude; reply ? msg
}
```

Reply Channels - Single Server

```
mtype = { nice, rude };
chan request = [0] of { mtype };
chan reply = [0] of { mtype };

active proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}

active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}

active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Is the assertion valid?

Reply Channels - Single Server

```
mtype = { nice, rude };
chan request = [0] of { mtype };
chan reply = [0] of { mtype };

active proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
  od
}

active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}

active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Is the assertion valid? Ask SPIN.

Several Servers

More realistic with several servers:

```
active [2] proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Several Servers

More realistic with several servers:

```
active [2] proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

Several Servers

More realistic with several servers:

```
active [2] proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

And here?

Several Servers

More realistic with several servers:

```
active [2] proctype Server() {
  mtype msg;
  do :: request ? msg; reply ! msg
od
}
active proctype NiceClient() {
  mtype msg;
  request ! nice; reply ? msg;
  assert(msg == nice)
}
active proctype RudeClient() {
  mtype msg;
  request ! rude; reply ? msg
}
```

And here? Analyse with SPIN.

Sending Channels via Channels

To fix the protocol:

Sending Channels via Channels

To fix the protocol:

clients declare local reply channel + send it to server

Sending Channels via Channels

```
mtype = { nice, rude };
chan request = [0] of { mtype, chan };

active [2] proctype Server() {
    mtype msg; chan ch;
    do :: request ? msg, ch;
        ch ! msg
    od
}

active proctype NiceClient() {
    chan reply = [0] of { mtype }; mtype msg;
    request ! nice, reply;    reply ? msg;
    assert( msg == nice )
}

active proctype RudeClient() {
    chan reply = [0] of { mtype }; mtype msg;
    request ! rude, reply;    reply ? msg
}
```

Sending Channels via Channels

```
mtype = { nice, rude };
chan request = [0] of { mtype, chan };

active [2] proctype Server() {
  mtype msg; chan ch;
  do :: request ? msg, ch;
    ch ! msg
  od
}

active proctype NiceClient() {
  chan reply = [0] of { mtype }; mtype msg;
  request ! nice, reply; reply ? msg;
  assert( msg == nice )
}

active proctype RudeClient() {
  chan reply = [0] of { mtype }; mtype msg;
  request ! rude, reply; reply ? msg
}
```

verify with SPIN

Scope of Channels

Global channel

- ▶ All processes can send and/or receive messages

Local channel

- ▶ Dies with its process
- ▶ Can model 'private' communication & security issues
- ▶ Example:
Local channel can be passed through a global channel

Sending Process IDs

Used *fixed constants* used for identification (here nice, rude)

Sending Process IDs

Used *fixed constants* used for identification (here nice, rude)

- ▶ inflexible
- ▶ doesn't scale

Sending Process IDs

Used *fixed constants* used for identification (here nice, rude)

- ▶ inflexible
- ▶ doesn't scale

Alternative:

Processes send their own, unique **process ID**, `_pid`, as part of message

Sending Process IDs

Used *fixed constants* used for identification (here nice, rude)

- ▶ inflexible
- ▶ doesn't scale

Alternative:

Processes send their own, unique **process ID**, `_pid`, as part of message

Experiment with `rendezvous3.pml`

Sending Process IDs

Used *fixed constants* used for identification (here nice, rude)

- ▶ inflexible
- ▶ doesn't scale

Alternative:

Processes send their own, unique **process ID**, `_pid`, as part of message

Experiment with `rendezvous3.pml`

Example, clients code:

```
chan reply = [0] of { byte, byte };
request ! _pid, reply;
reply ? serverID, serversClient;
```

Sending Process IDs

Used *fixed constants* used for identification (here nice, rude)

- ▶ inflexible
- ▶ doesn't scale

Alternative:

Processes send their own, unique **process ID**, `_pid`, as part of message

Experiment with `rendezvous3.pml`

Example, clients code:

```
chan reply = [0] of { byte, byte };
request ! _pid, reply;
reply ? serverID, serversClient;

assert( serversClient == _pid )
```

Limitations of Rendezvous Channels

- ▶ Rendezvous too restrictive for many applications
- ▶ Servers and clients block each other too much
- ▶ Difficult to manage uneven workload
(online shop: dozens of webserver serve thousands of clients)

Buffered Channel

Buffered channels queue messages.
Requests/services do not immediately block clients/servers.

Example:

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

Buffered Channels

Buffered channels, with capacity cap

- ▶ Can hold up to cap messages

Buffered Channels

Buffered channels, with capacity cap

- ▶ Can hold up to cap messages
- ▶ Are a FIFO (first-in-first-out) data structure:
always the 'oldest' message in channel is retrieved by a receive

Buffered Channels

Buffered channels, with capacity cap

- ▶ Can hold up to cap messages
- ▶ Are a FIFO (first-in-first-out) data structure:
always the 'oldest' message in channel is retrieved by a receive
- ▶ (Normal) receive statement reads **and** removes message

Buffered Channels

Buffered channels, with capacity cap

- ▶ Can hold up to cap messages
- ▶ Are a FIFO (first-in-first-out) data structure:
always the 'oldest' message in channel is retrieved by a receive
- ▶ (Normal) receive statement reads **and** removes message
- ▶ Sending and receiving to/from buffered channels is asynchronous, i.e. interleaved

Executability of Buffered Channel operations

Given channel ch , with capacity cap , currently containing n messages

receive statement $ch ? msg$

is executable iff ch is not empty, i.e., $n > 0$

send statement $ch ! msg$

is executable iff there is still 'space' in the message queue,
i.e., $n < cap$

A non-executable receive or send statement will **block** until it is executable again

Executability of Buffered Channel operations

Given channel ch , with capacity cap , currently containing n messages

receive statement $ch ? msg$

is executable iff ch is not empty, i.e., $n > 0$

send statement $ch ! msg$

is executable iff there is still 'space' in the message queue,
i.e., $n < cap$

A non-executable receive or send statement will **block** until it is executable again

(With option `-m`, SPIN has a different send semantics:

Attempt to send to full channel doesn't block, but message gets lost.)

Checking Channel for Full/Empty

This can prevent unnecessary blocking:

Given channel `ch`:

`full(ch)` checks whether `ch` is full

`nfull(ch)` checks whether `ch` is not full

`empty(ch)` checks whether `ch` is empty

`nempty(ch)` checks whether `ch` is not empty

Illegal to negate those.

Avoid combining with `else`.

Copy Message without Removing

Assume `ch` to be a buffered channel.

`ch ? color, time, flash`

- ▶ Assigns values from the message to `color`, `time`, `flash`
- ▶ Removes message from `ch`

Copy Message without Removing

Assume `ch` to be a buffered channel.

`ch ? color, time, flash`

- ▶ Assigns values from the message to `color`, `time`, `flash`
- ▶ Removes message from `ch`

`ch ? <color, time, flash>`

- ▶ Assign values from the message to `color`, `time`, `flash`
- ▶ Leaves message in `ch`

Dispatching Messages

Recurring task: Dispatch action depending on message

Dispatching Messages

Recurring task: Dispatch action depending on message

```
mtype = {hi, bye};
chan ch = [0] of {mtype};

active proctype Server () {
    mtype msg;
read:
    ch ? msg;
    do
        :: msg == hi -> printf("Hello.\n"); goto read
        :: msg == bye -> printf("See you.\n"); break
    od
}
...
```


Dispatching Messages

Recurring task: Dispatch action depending on message

```
mtype = {hi, bye};
chan ch = [0] of {mtype};

active proctype Server () {
    mtype msg;
read:
    ch ? msg;
    do
        :: msg == hi -> printf("Hello.\n"); goto read
        :: msg == bye -> printf("See you.\n"); break
    od
}
...
```

There is a better way!

Dispatching Messages

Recurring task: Dispatch action depending on message type.

```
mtype = {hi, bye};  
chan ch = [0] of {mtype};  
  
active proctype Server () {  
  do  
    :: ch ? hi -> printf("Hello.\n")  
    :: ch ? bye -> printf("See you.\n"); break  
  od  
}  
...
```

hi and bye are *values*, not variables!

Pattern Matching

Receive statement allows also non-variable expressions as arguments:

$$ch ? exp_1, \dots, exp_n$$

- ▶ exp_1, \dots, exp_n any(!) expressions of correct type
- ▶ Receive statement is **executable**, iff
 1. either
 - ▶ ch is buffered channel and not empty, or
 - ▶ ch is rendezvous channel and some process ready to send to ch
 2. message msg_1, \dots, msg_n in channel ch **matches** exp_1, \dots, exp_n
- ▶ msg_i **matches** exp_i iff
 - ▶ exp_i is a variable and msg_i a value (of correct type)
 - ▶ exp_i is not a variable and $exp_i == msg_i$

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5]` ?

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

▶ `[0, 5] ?` ✓

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?`

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?`

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

To match the **value** stored in a variable `var` use `eval(var)`

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

To match the **value** stored in a variable `var` use `eval(var)`

Does `ch ? 0, eval(id)` match message

- ▶ `[0, 5] ?`

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

To match the **value** stored in a variable `var` use `eval(var)`

Does `ch ? 0, eval(id)` match message

- ▶ `[0, 5] ?` ✓

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

To match the **value** stored in a variable *var* use `eval(var)`

Does `ch ? 0, eval(id)` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?`

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

To match the **value** stored in a variable `var` use `eval(var)`

Does `ch ? 0, eval(id)` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✗

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

To match the **value** stored in a variable *var* use `eval(var)`

Does `ch ? 0, eval(id)` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✗ `[1, 7] ?`

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

To match the **value** stored in a variable *var* use `eval(var)`

Does `ch ? 0, eval(id)` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✗ `[1, 7] ?` ✗

Pattern Matching Examples

Assume

```
chan ch = [0] of {int, int};  
int id = 5;
```

Does `ch ? 0, id` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✓ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

To match the **value** stored in a variable `var` use `eval(var)`

Does `ch ? 0, eval(id)` match message

- ▶ `[0, 5] ?` ✓ `[0, 7] ?` ✗ `[1, 7] ?` ✗
- ▶ Value of `id` afterwards?

Dispatching Messages Revisited

Random receive ?? (for buffered channels)

- ▶ Executable if matching message exists in channel.
- ▶ If executed, **first** matching message removed from channel.

Dispatching Messages Revisited

Random receive ?? (for buffered channels)

- ▶ Executable if matching message exists in channel.
- ▶ If executed, **first** matching message removed from channel.

```
mtype = {hi, bye};  
chan ch = [3] of {mtype};  
  
active proctype Server () {  
  do  
    :: ch ?? bye -> printf("See you.\n"); break  
    :: else      -> printf("Hello.\n")  
  od  
}  
...
```

Nicer Message Formatting

PROMELA provides an alternative, but equivalent syntax for

```
ch ! exp1, exp2, exp3
```

Nicer Message Formatting

PROMELA provides an alternative, but equivalent syntax for

```
ch ! exp1, exp2, exp3
```

namely

```
ch ! exp1(exp2, exp3)
```

Increases readability for certain applications, e.g. protocol modelling:

```
ch!send(msg,id) vs. ch!send,msg,id  
ch!ack(id) vs. ch!ack,id
```

And finally

Buffered channels are part of the state!

State space gets much bigger using buffered channels

Use with care (and with small buffers).