

Software Engineering using Formal Methods

Modeling Distributed Systems

Wolfgang Ahrendt & Wojciech Mostowski & Richard Bubel

13 September 2011

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** connected by
- ▶ **communication channels**
- ▶ **protocols** control data flow among nodes

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 the processes

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>	PROMELA data types

Example:

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

Meaning of Channels

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

Creates a channel, a **pointer** to which is stored in *name*

Meaning of Channels

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

Creates a channel, a **pointer** to which is stored in *name*

Messages communicated via the channel are *n*-tuples $\in \textcolor{red}{type_1} \times \dots \times \textcolor{red}{type_n}$

Meaning of Channels

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

Creates a channel, a *pointer* to which is stored in *name*

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

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

$\Rightarrow$ “*buffered channel*”

Meaning of Channels

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

Creates a channel, a **pointer** to which is stored in *name*

Messages communicated via the 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 a channel, a pointer to which is stored in `ch`

Meaning of Channels

Example:

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

Creates a channel, a pointer to which is stored in `ch`

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

Meaning of Channels

Example:

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

Creates a channel, a pointer to which is stored in `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 can be:

Meaning of Channels

Example:

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

Creates a channel, a pointer to which is stored in `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 can be: `green, 20, false`

Meaning of Channels

Example:

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

Creates a channel, a pointer to which is stored in `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 can be: `green, 20, false`

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

Sending and Receiving

send statement has the form:

name ! expr₁, ... , 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 ! expr₁, ... , expr_n

- ▶ *name*: channel variable
- ▶ *expr₁, ... , 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, 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, 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, 20, false*

receive statement has the form:

name ? var₁, ... , var_n

- ▶ *name*: channel variable

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, 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

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, 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*

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, 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 and 1 to request

Client-Server

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

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

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

```
...
```

Client0 and Client1 send messages 0 and 1 to 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

Demo

```
rendezvous1  
random simulation  
  
run spin -a ...
```

Executability of receive Statement

request ? num

executable only if a message is **available** in channel request

Executability of receive Statement

`request ? num`

executable only if a message is **available** in channel `request`

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

Executability of receive Statement

`request ? num`

executable only if a message is **available** in channel `request`

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

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

interactive simulation

Rendezvous Channels

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

/* global 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 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 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. Sender process' location counter at send ("!"): *"offer to engage in rendezvous"*

Rendezvous are Synchronous

Either:

1. Sender process' location counter at send ("!"): *"offer to engage in rendezvous"*
2. Receiver process' location counter at receive ("?"): *"rendezvous can be accepted"*

Rendezvous are Synchronous

Either:

1. Sender process' location counter at send ("!"): *"offer to engage in rendezvous"*
2. Receiver process' location counter at receive ("?"): *"rendezvous can be accepted"*

or the other way round:

1. Receiver process' location counter at receive ("?"): *"offer to engage in rendezvous"*

Rendezvous are Synchronous

Either:

1. Sender process' location counter at send ("!"): *"offer to engage in rendezvous"*
2. Receiver process' location counter at receive ("?"): *"rendezvous can be accepted"*

or the other way round:

1. Receiver process' location counter at receive ("?"): *"offer to engage in rendezvous"*
2. Sender process' location counter at send ("!"): *"rendezvous can be accepted"*

Rendezvous are Synchronous

Either:

1. Sender process' location counter at send ("!"): *"offer to engage in rendezvous"*
2. Receiver process' location counter at receive ("?"): *"rendezvous can be accepted"*

or the other way round:

1. Receiver process' location counter at receive ("?"): *"offer to engage in rendezvous"*
2. Sender process' location counter at send ("!"): *"rendezvous can be accepted"*

in any cases:

location counter of **both** processes is incremented at once

Rendezvous are Synchronous

Either:

1. Sender process' location counter at send ("!"): *"offer to engage in rendezvous"*
2. Receiver process' location counter at receive ("?"): *"rendezvous can be accepted"*

or the other way round:

1. Receiver process' location counter at receive ("?"): *"offer to engage in rendezvous"*
2. Sender process' location counter at send ("!"): *"rendezvous can be accepted"*

in any cases:

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 reply = [0] of { bool };

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

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

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

Reply Channels

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

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

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

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

(anonymous variable “_” used if interested in receipt, not content)

Reply Channels

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

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

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

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

But: client might get 'wrong' reply

Reply Channels - Single Server

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

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

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

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

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

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

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

Is the assertion valid?

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

Reply Channels - Single Server

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

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

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

Is the assertion valid? Ask SPIN.

```
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;  
  
}  
  
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

One way to fix the protocol:

clients declare local reply channel + send it to server

Sending Channels via Channels

One way to fix the protocol:

clients declare local reply channel + send it to server

(live in lecture)

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

channels are typically declared global

global channel

- ▶ usual case
- ▶ all processes can send and/or receive messages

local channel

- ▶ rarely used
- ▶ dies with its process
- ▶ can be useful to model security issues

example:

pointer to local channel could 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

example, clients code:

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

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

example, clients code:

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

assert( clientID == _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 no 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 from *cap*

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 from *cap*
- ▶ 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., $n < cap$

An 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., $n < cap$

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

(There is a SPIN option, $-m$, for a different send semantics: attempting to send to a full channel does not block, but the message gets lost instead.)

Checking Channel for Full/Empty

this can save from 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

with

cs ? color, time, flash

you

- ▶ assign values from the message to color, time, flash
- ▶ remove message from ch

Copy Message without Removing

with

cs ? color, time, flash

you

- ▶ assign values from the message to color, time, flash
- ▶ remove message from ch

with

cs ? <color, time, flash>

you

- ▶ assign values from the message to color, time, flash
- ▶ **leave** message in ch

Dispatching Messages

Recurring task: Dispatch action depending on message type.

Dispatching Messages

Recurring task: Dispatch action depending on message type.

```
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 type.

```
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
}
...
```

Boring to write ..., but there is a better way!

Pattern Matching

Receive statement allows also values as arguments:

$$cs \ ? \ exp_1, \dots, exp_n$$

- ▶ $exp_1, \dots, exp_n$ any(!) expressions of correct type
- ▶ statement is **executable**, if message $msg_1, \dots, msg_n$ in channel cs **matches** arguments, i.e. if
 - ▶ exp_i is a value, e.g. 23, msg_i must have same value
 - ▶ exp_i is a variable, then any value of msg_i (of correct type) matches and is assigned if statement is executed

Pattern Matching

Receive statement allows also values as arguments:

$$cs \text{ ? } exp_1, \dots, exp_n$$

- ▶ $exp_1, \dots, exp_n$ any(!) expressions of correct type
- ▶ statement is **executable**, if message $msg_1, \dots, msg_n$ in channel cs **matches** arguments, i.e. if
 - ▶ exp_i is a value, e.g. 23, msg_i must have same value
 - ▶ exp_i is a variable, then any value of msg_i (of correct type) matches and is assigned if statement is executed

Example

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

Pattern Matching

Receive statement allows also values as arguments:

$$cs \text{ ? } exp_1, \dots, exp_n$$

- ▶ $exp_1, \dots, exp_n$ any(!) expressions of correct type
- ▶ statement is **executable**, if message $msg_1, \dots, msg_n$ in channel cs **matches** arguments, i.e. if
 - ▶ exp_i is a value, e.g. 23, msg_i must have same value
 - ▶ exp_i is a variable, then any value of msg_i (of correct type) matches and is assigned if statement is executed

Example

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

Does $cs \text{ ? } 0, id$ match message

- ▶ $[0, 5] \text{ ?}$

Pattern Matching

Receive statement allows also values as arguments:

$$cs \text{ ? } exp_1, \dots, exp_n$$

- ▶ $exp_1, \dots, exp_n$ any(!) expressions of correct type
- ▶ statement is **executable**, if message $msg_1, \dots, msg_n$ in channel cs **matches** arguments, i.e. if
 - ▶ exp_i is a value, e.g. 23, msg_i must have same value
 - ▶ exp_i is a variable, then any value of msg_i (of correct type) matches and is assigned if statement is executed

Example

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

Does $cs \text{ ? } 0, id$ match message

- ▶ $[0, 5] \text{ ? } \checkmark$

Pattern Matching

Receive statement allows also values as arguments:

$$cs \text{ ? } exp_1, \dots, exp_n$$

- ▶ $exp_1, \dots, exp_n$ any(!) expressions of correct type
- ▶ statement is **executable**, if message $msg_1, \dots, msg_n$ in channel cs **matches** arguments, i.e. if
 - ▶ exp_i is a value, e.g. 23, msg_i must have same value
 - ▶ exp_i is a variable, then any value of msg_i (of correct type) matches and is assigned if statement is executed

Example

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

Does $cs \text{ ? } 0, id$ match message

- ▶ $[0, 5] \text{ ? } \checkmark$ $[0, 7] \text{ ? }$

Pattern Matching

Receive statement allows also values as arguments:

$$cs \text{ ? } exp_1, \dots, exp_n$$

- ▶ $exp_1, \dots, exp_n$ any(!) expressions of correct type
- ▶ statement is **executable**, if message $msg_1, \dots, msg_n$ in channel cs **matches** arguments, i.e. if
 - ▶ exp_i is a value, e.g. 23, msg_i must have same value
 - ▶ exp_i is a variable, then any value of msg_i (of correct type) matches and is assigned if statement is executed

Example

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

Does $cs \text{ ? } 0, id$ match message

- ▶ $[0, 5] \text{ ? } \checkmark$ $[0, 7] \text{ ? } \checkmark$
- ▶ $[1, 7] \text{ ? }$

Pattern Matching

Receive statement allows also values as arguments:

$$cs \text{ ? } exp_1, \dots, exp_n$$

- ▶ $exp_1, \dots, exp_n$ any(!) expressions of correct type
- ▶ statement is **executable**, if message $msg_1, \dots, msg_n$ in channel cs **matches** arguments, i.e. if
 - ▶ exp_i is a value, e.g. 23, msg_i must have same value
 - ▶ exp_i is a variable, then any value of msg_i (of correct type) matches and is assigned if statement is executed

Example

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

Does $cs \text{ ? } 0, id$ match message

- ▶ $[0, 5] \text{ ? } \checkmark$ $[0, 7] \text{ ? } \checkmark$
- ▶ $[1, 7] \text{ ? } \times$

Pattern Matching

Receive statement allows also values as arguments:

$$cs \ ? \ exp_1, \dots, exp_n$$

- ▶ $exp_1, \dots, exp_n$ any(!) expressions of correct type
- ▶ statement is **executable**, if message $msg_1, \dots, msg_n$ in channel cs **matches** arguments, i.e. if
 - ▶ exp_i is a value, e.g. 23, msg_i must have same value
 - ▶ exp_i is a variable, then any value of msg_i (of correct type) matches and is assigned if statement is executed

Example

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

Does $cs \ ? \ 0, id$ match message

- ▶ $[0, 5] \ ? \ \checkmark$ $[0, 7] \ ? \ \checkmark$
- ▶ $[1, 7] \ ? \ \times$

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

Dispatching Messages Revisited

Recurring task: Dispatch action depending on message type.

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

Dispatching Messages Revisited

Recurring task: Dispatch action depending on message type.

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

Random receive ?? (for buffered channels)

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

Nicer Message Formatting

PROMELA provides an alternative, but equivalent syntax for

```
cs ! exp1, exp2, exp3
```

Nicer Message Formatting

PROMELA provides an alternative, but equivalent syntax for

```
cs ! exp1, exp2, exp3
```

namely

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

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

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