Formal Specification and Verification

Modeling Distributed Systems

Bernhard Beckert

Based on a lecture by Wolfgang Ahrendt and Reiner Hähnle at Chalmers University, Göteborg

This Lecture

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** $\{type_1, \ldots, type_n\}$

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** $\{type_1, \ldots, type_n\}$

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

Meaning of Channels

chan *name* = [*capacity*] **of** $\{type_1, ..., type_n\}$

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

messages communicated via the channel are n-tuples $\in type_1 \times ... \times type_n$

Meaning of Channels

chan *name* = [*capacity*] **of** $\{type_1, ..., type_n\}$

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

messages communicated via the channel are *n*-tuples $\in type_1 \times ... \times type_n$

can buffer up to *capacity* messages, if $capacity \geq 1$
$\Rightarrow$ *"buffered channel"*

Meaning of Channels

$\mathbf{chan}$ *name* = [*capacity*] $\mathbf{of}$ $\{type_1, ..., type_n\}$

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

messages communicated via the channel are *n*-tuples $\in type_1 \times ... \times type_n$

can buffer up to *capacity* messages, if $capacity \geq 1$
$\Rightarrow$ *"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 $\in \mathbf{mtype} \times \mathbf{byte} \times \mathbf{bool}$

Meaning of Channels

example:

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

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

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

given, e.g., $\textbf{mtype}$ `{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 $\in$ **mtype** $\times$ **byte** $\times$ **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 $\in$ **mtype** $\times$ **byte** $\times$ **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:

$$\mathit{name}\ !\ \mathit{expr}_1, \ldots, \mathit{expr}_n$$

Sending and Receiving

send statement has the form:

$name\ !\ expr_1, \ldots, expr_n$

- *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 ! $expr_1$, ... , $expr_n$

- *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

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, 20, false`

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, 20, false`

receive statement has the form:

name ? var_1, ... , var_n

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, 20, false`

receive statement has the form:

name ? var_1, ... , 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, 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:

$name\ \texttt{!}\ expr_1, \ldots, expr_n$

- *name*: channel variable
- $expr_1, \ldots, expr_n$: sequence of expressions, where number and types match message type
- sends *values* of $expr_1, \ldots, expr_n$ as *one* message
- example: `ch ! green, 20, false`

receive statement has the form:

$name\ \texttt{?}\ var_1, \ldots, var_n$

- *name*: channel variable
- $var_1, \ldots, var_n$: sequence of variables, where number and types match message type
- *assigns* values of message to $var_1, \ldots, var_n$

Sending and Receiving

send statement has the form:

name `!` $expr_1, \ldots, expr_n$

- *name*: channel variable
- $expr_1, \ldots, expr_n$: sequence of expressions, where number and types match message type
- sends *values* of $expr_1, \ldots, expr_n$ as *one* message
- example: `ch ! green, 20,` **false**

receive statement has the form:

name `?` $var_1, \ldots, var_n$

- *name*: channel variable
- $var_1, \ldots, var_n$: sequence of variables, where number and types match message type
- *assigns* values of message to $var_1, \ldots, var_n$
- example: `ch ? color, time, flash`

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

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`

$\Rightarrow$ 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
```

Demo

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

Demo

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
$\vdots$		$\vdots$
`(11,45)`	$\longrightarrow$	`(hour,minute)`
$\vdots$		$\vdots$

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"

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

```
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
}
```

```
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
}
```

```
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
}
```

Is the assertion valid?

```
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
}
```

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
}
```

Is the assertion correct 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
}
```

Is the assertion correct here? analyze 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

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 webservers 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 safe 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`

And finally

Buffered channels are part of the state!

State space gets much bigger using buffered channels

Use with care (and with small buffers).