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
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
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:

\[
\text{chan } \text{name} = [\text{capacity}] \text{ of } \{\text{type}_1, \ldots, \text{type}_n\}
\]

- **name**: name of channel variable
- **capacity**: non-negative integer constant
- **type\textsubscript{i}**: PROMELA data types

Example:

\[
\text{chan } \text{ch} = [2] \text{ of } \{\text{mtype, byte, bool}\}
\]
Meaning of Channels

```
chan name = [capacity] of {type_1, ..., type_n}
```

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 \( \in type₁ \times ... \times typeₙ \)
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 \(\geq 1\)

⇒ “buffered channel”
Meaning of Channels

```plaintext
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”

the channel has *no* buffer, if \( capacity = 0 \)

\( \Rightarrow \) “rendezvous channel”
Meaning of Channels

example:

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

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

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

creates a channel, a pointer to which is stored in \text{ch}

messages communicated via \text{ch} are 3-tuples \( \in \text{mtype} \times \text{byte} \times \text{bool} \)
Meaning of Channels

example:

```go
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:

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

does 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: \( \text{green, 20, false} \)
Meaning of Channels

example:

```go
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 `∈ mtype × byte × 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.
send statement  has the form:

\[ name \! \ expr_1, \ldots, expr_n \]
Sending and Receiving

**send statement** has the form:

\[ \text{name} ! \ expr_1, \ldots, \ expr_n \]

- **name**: channel variable

**receive statement** has the form:

\[ \text{name} ? \ var_1, \ldots, \ var_n \]

- **name**: channel variable
- **var_1, \ldots, var_n**: sequence of variables

Assigns values of message to **var_1, \ldots, var_n**

**example:**

\[ \text{ch} ! \text{green}, 20, \text{false} \]
**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

**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
Sending and Receiving

**send statement** has the form:

```
name ! expr₁, ... , exprₙ
```

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

**receive statement** has the form:

```
name ? var₁, ... , varₙ
```

- **name**: channel variable
- **var₁, ... , varₙ**: sequence of variables, where number and types match message type
- assigns *values of* message to `var₁, ... , varₙ`
**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: \( ch \! \ green, 20, false \)
Send and Receiving

**send statement** has the form:

\[ \text{name} ! \ expr_1, \ldots, \ expr_n \]

- **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: \( ch ! \ \text{green}, \ 20, \ \text{false} \)

**receive statement** has the form:

\[ \text{name} ? \ var_1, \ldots, \ var_n \]
Sending and Receiving

**send statement** has the form:

\[
\text{name} ! \; \text{expr}_1, \ldots, \text{expr}_n
\]

- **name**: channel variable
- **expr\(_1\), \ldots, expr\(_n\)**: sequence of expressions, where number and types match message type
- sends values of \(\text{expr}_1, \ldots, \text{expr}_n\) as one message
- example: \(\text{ch} ! \; \text{green}, 20, \text{false}\)

**receive statement** has the form:

\[
\text{name} ? \; \text{var}_1, \ldots, \text{var}_n
\]

- **name**: channel variable
Sending and Receiving

**send statement** has the form:

\[
name \ ! \ expr_1, \ldots, \ expr_n
\]

- \textit{name:} channel variable
- \textit{expr}_1, \ldots, \textit{expr}_n: sequence of expressions, where number and types match message type
- sends \textit{values} of \textit{expr}_1, \ldots, \textit{expr}_n as one message
- example: \texttt{ch ! green, 20, false}

**receive statement** has the form:

\[
name \ ? \ var_1, \ldots, \ var_n
\]

- \textit{name:} channel variable
- \textit{var}_1, \ldots, \textit{var}_n: sequence of variables, where number and types match message type
**Sending and Receiving**

**send statement** has the form:

\[ \text{name} \!\! \text{expr}_1, \ldots, \text{expr}_n \]

- **name**: channel variable
- \( \text{expr}_1, \ldots, \text{expr}_n \): sequence of expressions, where number and types match message type
- sends values of \( \text{expr}_1, \ldots, \text{expr}_n \) as one message
- example: \text{ch ! green, 20, false}

**receive statement** has the form:

\[ \text{name} \? \text{var}_1, \ldots, \text{var}_n \]

- **name**: channel variable
- \( \text{var}_1, \ldots, \text{var}_n \): sequence of variables, where number and types match message type
- assigns values of message to \( \text{var}_1, \ldots, \text{var}_n \)

---

Formal Specification and Verification: Distributed Systems
B. Beckert 9 / 45
**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
channel request = [0] of { byte };

active proctype Client0() {
    request ! 0;
}

active proctype Client1() {
    request ! 1;
}

...
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

```haskell
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

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

...

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

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

...  

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

Server loops on:
```
Client-Server

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

...

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

Server loops on:

- receiving first message from request,
Client-Server

```c
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

```go
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

rendezvous 1
random simulation

run spin -a ...

Formal Specification and Verification: Distributed Systems
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
Demo

interactive simulation
Rendezvous Channels

```plaintext
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

```plaintext
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? ⇒ 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 \hspace{1cm} Receiver
(11,45) \hspace{1cm} (\text{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”
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
chan request = [0] of { byte };

active proctype Server() {
    byte num;
    do :: request ? num ->
        printf("serving \client<num>\d\n", num)
    od
}
active proctype Client0() {
    request ! 0
}
active proctype Client1() {
    request ! 1
}
Reconsider Client Server

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

active proctype Server() {
    byte num;
    do :: request ? num ->
        printf("serving\nclient\n%d
", num)
    od
}
active proctype Client0() {
    request ! 0
}
active proctype Client1() {
    request ! 1
}

so far no reply to clients
```
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 ? _
}
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)
chan request = [0] of { byte };  
chan reply = [0] of { bool };

active proctype Server() {
    byte num;
    do :: request ? num ->
        printf("serving\text{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.
More realistic with several servers:

```plaintext
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
}
```
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
}
```
More realistic with several servers:

```plaintext
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:

```plaintext
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

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

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

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

\[
\text{active proctype RudeClient()} \{ \\
    \text{chan reply} = [0] \text{ of } \{ \text{mtype} \}; \text{ mtype msg;} \\
    \text{request ! rude, reply; reply ? msg}
\}
\]
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 \textit{fixed constants} used for identification (here \texttt{nice}, \texttt{rude})

- inflexible
- doesn’t scale

Alternative:
processes send their own, unique \texttt{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:

```go
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:

```go
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:
```go
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 \textit{and} removes message from $cap$
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
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.
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 $\texttt{SPIN}$ 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 \( ch \):

\( \text{full}(ch) \) checks whether \( ch \) is full
\( \text{nfull}(ch) \) checks whether \( ch \) is not full
\( \text{empty}(ch) \) checks whether \( ch \) is empty
\( \text{nempty}(ch) \) 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
with

\texttt{cs ? color, time, flash}

you

- assign values from the message to color, time, flash
- remove message from \texttt{ch}

with

\texttt{cs ? <color, time, flash>}

you

- assign values from the message to color, time, flash
- \texttt{leave} message in \texttt{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).