Formale Methoden im Software Entwurf

Modellierung verteilter Systeme / Modeling Distributed Systems

Reiner Hähnle

12 November 2018

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*

Google went down today. The Internet went bananas.

By Zee, Saturday, 17 Aug '13 , 01:15am

Using PROMELA channels for modeling distributed systems

Modeling Distributed Systems

Distributed systems consist of

- nodes connected by
- communication channels with
- protocols controlling the data flow among nodes

Distributed systems are very complex

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

PROMELA Model of Distributed Systems

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

(Rendezvous) 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 this initialization:

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, name is a reference to it

Messages communicated via the channel are tuples $\in type_1 \times \ldots \times type_n$

Channel 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

Example

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

- Creates a channel, a reference to which is stored in `ch`
- Messages communicated via `ch` are triples $\in$ **mtype** $\times$ **byte** $\times$ **bool**
- Given, e.g., **mtype** `{red, yellow, green}`,
  an example message might be: (`green, 20,` **false**)
- `ch` is a buffered channel, buffering up to 2 messages

Sending and Receiving

Send statement has the form:

name ! $expr_1, \ldots, expr_n$

- name: channel variable
- $expr_1, \ldots, expr_n$: sequence of expressions
  whose number, types match declaration of channel name
- Sends values of $expr_1, \ldots, expr_n$ as a single 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
  whose number, types match declaration of channel name
- 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
- Standard case
- All processes can send and/or receive messages

Local channel
- Less often used
- Dies with its process
- Can be useful to model security issues
  - reference to local channel may be passed through a global channel

Client-Server Model with Channels (Client Side)

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

active proctype Client0() {
  request ! 0;
}

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

`Client0` and `Client1` send messages 0 and 1 to channel `request`

Order of sending is non-deterministic

Client-Server Model with Channels (Server Side)

```
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 received value

Demo

`spin rendezvous1.pml`

- Random simulation
- Note the invalid end states
  - Verification attempt of `rendezvous1` will indicate deadlock
  - Server cannot proceed $\Rightarrow$ executability of receive statement

Executability of Send/Receive Statement

statement type	executable
assignments	always
assertions	always
print statements	always
expression statements	iff value not 0/**false**

`name ! msg`	iff some process wants to receive on `name`
`name ? msg`	iff a message available in channel `name`

Receive statement frequently used as guard in **if**/**do**-statements

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

Demo

```
spin -i rendezvous1
```

- Receive statement only available after first request sent
- Why no more interactive choices immediately after first send?

Interleaving of Rendezvous Channels

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


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

active proctype Receiver() {
  byte hour, minute;

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

Which interleavings can occur?

Demo

`ReadySteady.pml`

- Interactive simulation `spin -i ReadySteady.pml`
- After selecting `Sender`, instruction pointer of `Sender` is at **printf**`("Sent␣n")` and instruction pointer of `Receiver` is at **printf**`("Received␣n")`

Less interleavings than perhaps expected!

Rendezvous are Synchronous

The following holds for all rendezvous channels:

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

Sender		Receiver
⋮		⋮
`(11,45)`	$\longrightarrow$	`(hour,minute)`
⋮		⋮

Rendezvous are Synchronous (Cont'd)

Either sender arrives first at rendezvouz

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 receiver arrives first at rendezvouz

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"

- Either way, location counter of both processes 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
}
```

So far: no reply to clients — not very useful!

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 when interested in receipt, not content)

Reply Channels Cont'd (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
}
```

Is the assertion valid? Ask SPIN *rude1.pml*

Reply Channels Cont'd (Multiple Servers)

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

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
}
```

Analyse with SPIN: rude2.pml

Sending Channels via Channels

One way to fix the protocol:

- Clients declare local reply channel + send it to server
- Situation where local channels are useful

Demo `rude3.pml`, see code next slide

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 Process IDs

Examples use fixed constants for client identification (here `nice`, `rude`)

- Inflexible
- Brittle code (changes require consistent renaming)
- Doesn't scale to sets of clients

Improvement:

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

Example (Client Code)

```
byte serverID, clientID;
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: several webservers serve thousands of clients)

Buffered Channels

Buffered channels queue messages:
requests/services do not immediately block clients/servers

Example (Declaration of buffered channel with capacity 3)

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

Buffered Channels Cont'd

Buffered channels with capacity cap

- Can hold up to cap messages
- Are a FIFO (first-in-first-out) data structure:
  the "oldest" message in a channel is retrieved by receive
- Receive statement by default reads and removes message
- Sending and Receiving to/from buffered channels is asynchronous:
  interleaving may occur between sending and receiving

Executability of Buffered Channel Operations

Given channel `name` with capacity *cap*, currently containing *n* messages

statement type	executable
assignments	always
assertions	always
print statements	always
expression statements	iff value not $0/\mathbf{false}$

`name ! msg`	iff message queue is not full, i.e. $n < cap$
`name ? msg`	iff channel `name` is not empty, i.e. $n > 0$

- Non-executable receive/send statements block until they become executable
- There is a Spin option, `-m`, for a different send semantics: send to a full channel does not block, but the message is lost instead

Checking Channels for Being Full/Empty

Polling guards for full/empty channels prevent unwanted 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

- Cannot negate these guards
- Avoid combining with **else**
  - **else** is implicit negation of remaining guards
  - Results in unintuitive blocking behavior
- For the same reason, avoid combining send statement with **else**

Copy A Message Without Removing It

Syntax for receiving message without removing it from channel

```
ch ?  <v1, ..., vN>
```

- where `v1,...,vN` are variables to which channel value assigned

Example

```
cs ? color, time, flash
```

- assigns values from the message to `color, time, flash`
- removes message from `ch`

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

- assigns values from the message to `color, time, flash`
- leaves message in `ch`

Dispatching Messages

A Frequently Recurring Task

Dispatch action depending on message type:

```
mtype = {hello, goodbye};
chan ch = [0] of {mtype};

active proctype Server () {
  mtype msg;
read:
  ch ? msg;
  do
    ::  msg == hello    -> printf("Hi\n"); goto read
    ::  msg == goodbye -> printf("Bye\n"); break
  od
}
```

Clumsy code . . . but there is a better way!

Pattern Matching

Pattern in Receive Statement

- Receive statement admits values as arguments:

$$ch \ ? \ exp_1, \ldots, exp_n$$

- Each exp_i is either a variable or a value
- Types of $exp_1, \ldots, exp_n$ must comply to type of ch
- Each exp_i is matched against the message msg_i returned from ch
  - If exp_i is value then $exp_i = msg_i$ must hold
  - If exp_i is variable complying to type of msg_i then assign msg_i to exp_i
  - Otherwise, matching fails
- Receive statement is executable iff matching succeeds

Pattern Matching Example

Example

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

Does `cs ? 0, id` match message

- `[0, 5]` ? ✔ `[0, 7]` ? ✔ (value of `id` is now 7)
- `[1, 7]` ? ✘

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

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

- `[0, 7]` ? ✘

Dispatching Messages By Pattern Matching

```
mtype = {hello, goodbye};
chan ch = [0] of {mtype};

active proctype Server () {
  do /* goto not needed anymore!  */
    :: ch ? hello   -> printf("Hi\n")
    :: ch ? goodbye -> printf("Bye\n"); break
  od
}
```

Concise programming idiom for message dispatch

Random Receive

Random Receive Statement

For buffered channels can use syntax ch ?? $exp_1, \ldots, exp_n$

- Executable iff a matching message exists somewhere in channel
- Any, not only the first, message in channel buffer is matchable
- If executed, first matching message removed from channel
- Use to transmit messages with different purposes in one channel
- Name "random receive" is confusing—outcome is deterministic!

Prefix Syntax for Messages

PROMELA provides an alternative, but equivalent syntax for

```
cs ! exp1, exp2, ..., expN
```

namely

```
cs ! exp1(exp2, ..., expN)
```

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

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

Buffered Channels and Verification

State space to be traversed during verification
much increased by buffered channels

Keep In Mind

- Buffered channels are part of the state
- Don't use buffered channels unless they are needed
- Set capacity of buffered channels as low as possible
- Make channel types as small as possible
(holds even for rendezvous channels)

Literature for this Lecture

Ben-Ari Chapter 7