Proving Copyless Message Passing

Jules Villard[1] Étienne Lozes[1] Cristiano Calcagno[2]

[1]LSV, ENS Cachan, CNRS

[2]Imperial College, London

APLAS'09 Conference
15 December 2009

Outline

Introduction

Programming Language

Contracts

Separation Logic

Conclusion

Language's Specifications

We want to model programs with the following features:

- Explicit memory manipulation (no garbage collection)
- Copyless, asynchronous message passing
  - Instead of copying the contents of the message, send a pointer to it and transfer ownership
  - Assumes a shared memory

What to Prove

We are interested in the following properties:

- no memory fault
- no races
- no memory leaks
- safe communications

Proving Copyless Message Passing Programs

- We mix separation logic and contracts
  - separation logic gives us safety properties
  - contracts give us liveness properties
- the combination of the two gives us something more than the two separately (*e.g.* no memleaks)

Outline

Communication Model

- Channels are bidirectional and asynchronous

  channel = pair of FIFO queues
- Channels are made of two endpoints

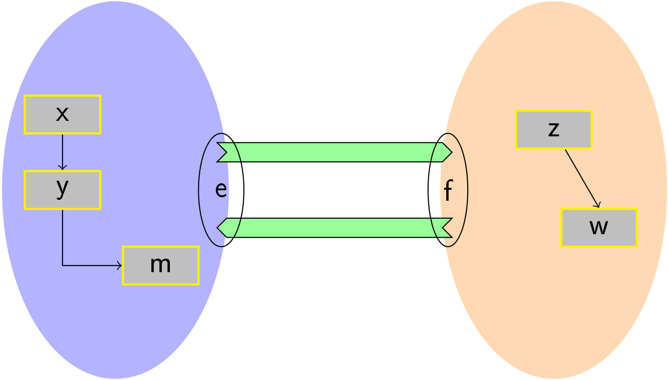
  similar to the socket model
- Endpoints can be allocated, disposed of, and communicated through channels

  similar to the π-calculus
- Communications are ruled by user-defined contracts

  similar to session types
- Inspired by Sing#, the language of the Singularity OS

  [Fähndrich & al. '06]

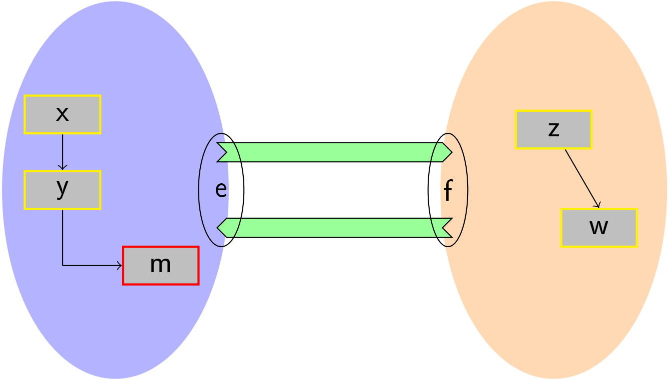
Message Passing with copies



`send(cell,e,m);`

`z→left = receive(cell,f);`

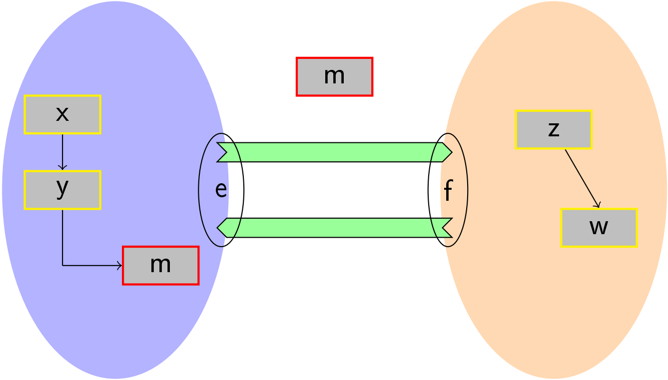
Message Passing with copies



`send(cell,e,m);`

`z→left = receive(cell,f);`

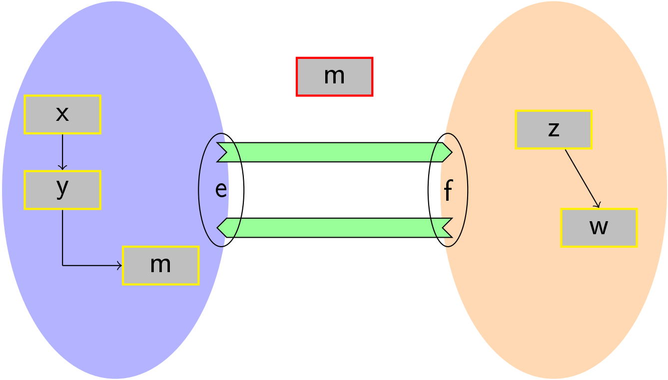
Message Passing with copies



`send(cell,e,m);`

`z→left = receive(cell,f);`

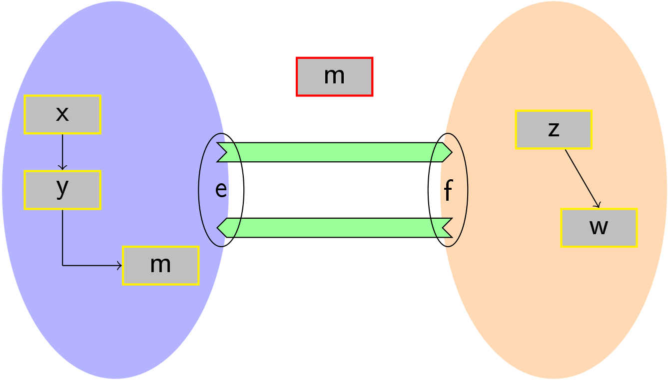
Message Passing with copies



`send(cell,e,m);`

`z→left = receive(cell,f);`

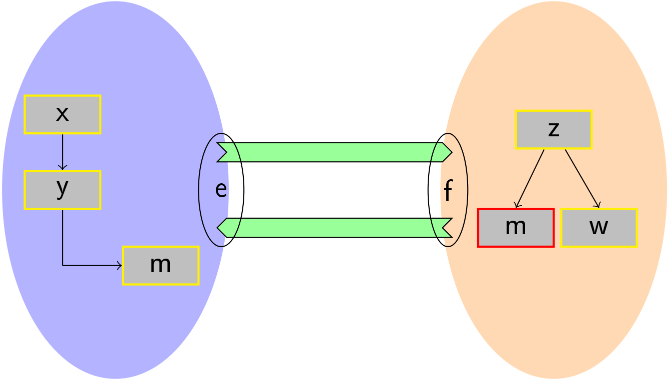
Message Passing with copies




```
send(cell,e,m);
```

```
z→left = receive(cell,f);
```

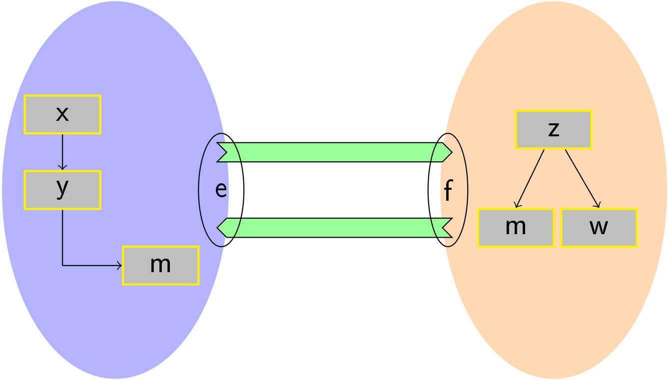
Message Passing with copies



`send(cell,e,m);`

`z→left = receive(cell,f);`

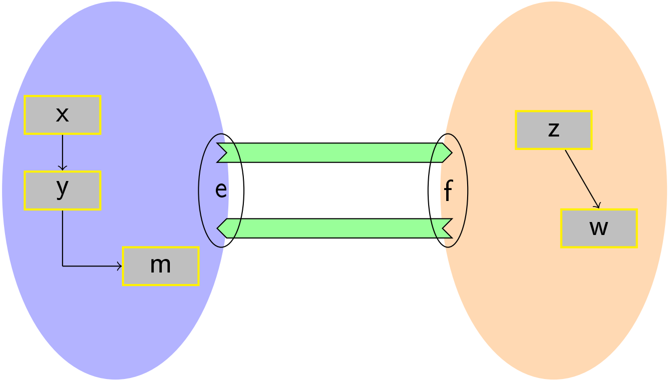
Message Passing with copies



`send(cell,e,m);`

`z→left = receive(cell,f);`

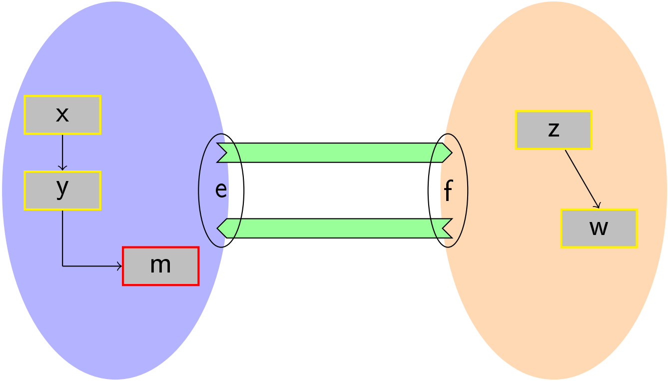
Copyless Message Passing (shared memory)




```
send(cell,e,m);
```

```
z→left = receive(cell,f);
```

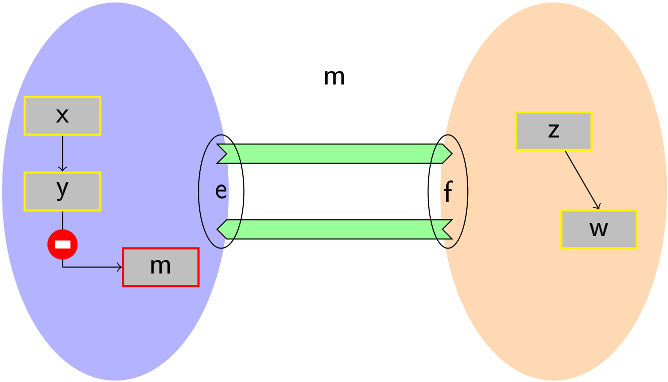
Copyless Message Passing (shared memory)



`send(cell,e,m);`

`z→left = receive(cell,f);`

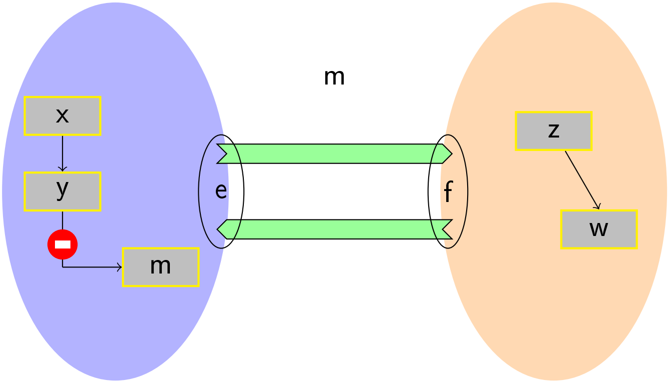
Copyless Message Passing (shared memory)



`send(cell,e,m);`

`z→left = receive(cell,f);`

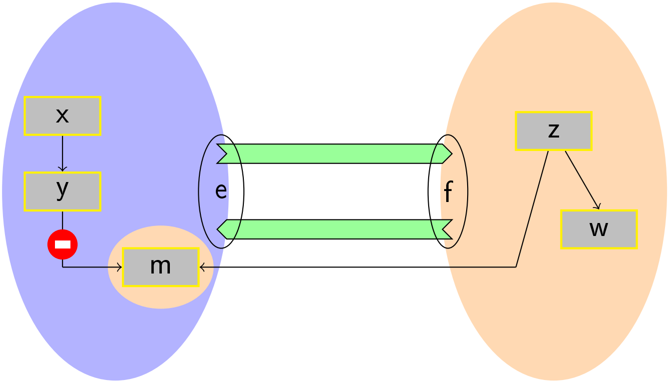
Copyless Message Passing (shared memory)



`send(cell,e,m);`

`z→left = receive(cell,f);`

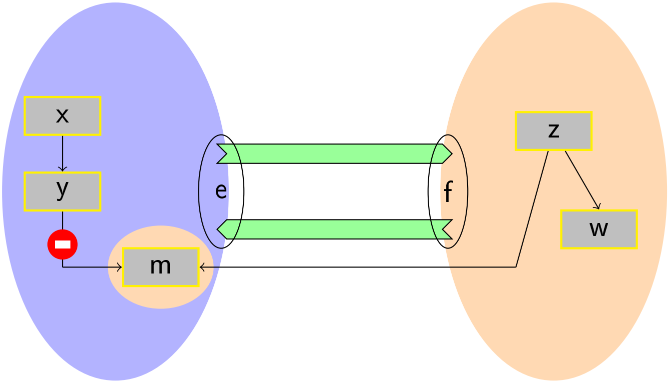
Copyless Message Passing (shared memory)



`send(cell,e,m);`

`z→left = receive(cell,f);`

Copyless Message Passing (shared memory)



`send(cell,e,m);`

`z→left = receive(cell,f);`

Example

```
message cell

put_get() {
  local e,f,x;
  (e,f) = open(C);
  x = new();
  put(e,x) || get(f);
  close(e,f);
}
```

```
put(e,x) {
  send(cell,e,x);
}


get(f) {
  local y;
  y = receive(cell,f);
  dispose(y);
}
```

Outline

Introduction

Programming Language

Contracts

Separation Logic

Conclusion

Contracts dictate which sequences of messages are admissible.

- It is a finite state machine, whose arrows are labeled by a message's name and a direction: send (!) or receive (?).
- Dual endpoints of a channel follow dual contracts ($\bar{C} = C[? \leftrightarrow !]$).

Contract of the Example

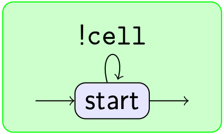
```
message cell
contract C {
  initial final state start
       { !cell -> start; }
}
```

```
put_get() {
  local e,f,x;
  (e,f) = open(C);
  x = new();
  put(e,x) || get(f);
  close(e,f);
}
```

```
put(e,x) {
  send(cell,e,x);
}
```

```
get(f) {
  local y;
  y = receive(cell,f);
  dispose(y);
}
```

C



Contract of the Example

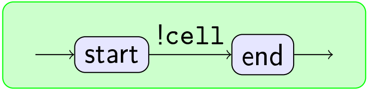
```
message cell
contract C {
  initial state start
        { !cell -> end; }
  final state end {}
}

put_get() {
  local e,f,x;
  (e,f) = open(C);
  x = new();
  put(e,x) || get(f);
  close(e,f);
}
```

```
put(e,x) {
  send(cell,e,x);
}

get(f) {
  local y;
  y = receive(cell,f);
  dispose(y);
}
```

C



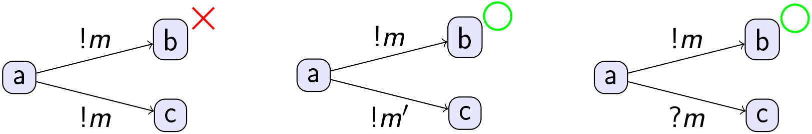
Leak-Free Contract

A contract is leak-free if for all communications, whenever two endpoints of a channel following the contract are in the same final state, then the message queues are empty.

- Determining whether a given contract is leak-free or not is undecidable.
- We rely on simple sufficient conditions for a contract to be leak-free.

Definition 1 (Determinism)

Two distinct edges in a contract must be labeled by different messages.

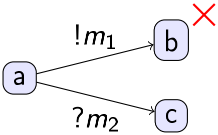


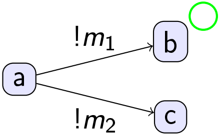
Properties of Contracts

Definition 1 (Determinism)

Definition 2 (Uniform choice)

All outgoing edges from a same state in a contract must be either all sends or all receives.





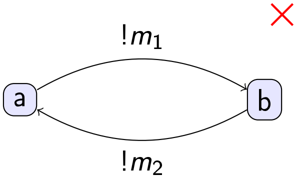
Properties of Contracts

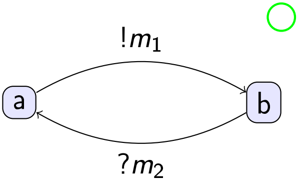
Definition 1 (Determinism)

Definition 2 (Uniform choice)

Definition 3 (Synchronizing state)

A state s is synchronizing if every cycle that goes through it contains at least one send and one receive.





Properties of Contracts

Definition 1 (Determinism)

Definition 2 (Uniform choice)

Definition 3 (Synchronizing state)

Lemma 4 (Half-Duplex)

1 & 2 $\Rightarrow$ *communications are half-duplex.*

Lemma 5 (Leak-free)

final states are synchronizing and communications are half-duplex $\Rightarrow$ *contract is leak-free*

Outline

Separation Logic

Separation Logic [Reynolds 02, O'Hearn 01, ...]

- An assertion language to describe states
- An extension of Hoare Logic

Syntax

$$
\begin{array}{lcll}
E & ::= & x \mid n \in \mathbb{N} & \text{expressions} \\
A & ::= & \quad E_1 = E_2 \mid E_1 \neq E_2 & \text{stack predicates} \\
 & & \mid \mathsf{emp}_h \mid E_1 \mapsto E_2 \mid \mathrm{list}(E) & \text{heap predicates} \\
 & & \mid \exists x.\, A \mid A_1 \wedge A_2 \mid \neg A \mid A_1 * A_2 & \text{formulas}
\end{array}
$$

Semantics

$$
\begin{array}{lll}
(s,h) \models \; E_1 = E_2 & \text{iff} & [\![E_1]\!]s = [\![E_2]\!]s \\
(s,h) \models \; \mathsf{emp}_h & \text{iff} & dom(h) = \emptyset \\
(s,h) \models \; E_1 \mapsto E_2 & \text{iff} & dom(h) = \{[\![E_1]\!]s\} \;\&\; h([\![E_1]\!]s) = [\![E_2]\!]s
\end{array}
$$

$$
\mathrm{list}(E) \triangleq (E = 0 \wedge \mathsf{emp}_h) \vee (\exists x.\, E \mapsto x * \mathrm{list}(x))
$$

Syntax

$$
\begin{array}{rcll}
E & ::= & x \mid n \in \mathbb{N} & \text{expressions} \\
A & ::= & \quad E_1 = E_2 \mid E_1 \neq E_2 & \text{stack predicates} \\
 & & \mid \mathsf{emp}_h \mid E_1 \mapsto E_2 \mid \mathrm{list}(E) & \text{heap predicates} \\
 & & \mid \exists x.\, A \mid A_1 \wedge A_2 \mid \neg A \mid A_1 * A_2 & \text{formulas}
\end{array}
$$

Semantics

$$
\begin{array}{rlll}
(s,h) \vDash & A_1 \wedge A_2 & \text{iff} & (s,h) \vDash A_1 \ \& \ (s,h) \vDash A_2 \\
(s,h) \vDash & \neg A & \text{iff} & (s,h) \nvDash A \\
(s,h) \vDash & A_1 * A_2 & \text{iff} & \exists h_1, h_2.\, dom(h_1) \cap dom(h_2) = \emptyset \\
 & & & \qquad \& \ h = h_1 \cup h_2 \\
 & & & \qquad \& \ (s,h_1) \vDash A_1 \ \& \ (s,h_2) \vDash A_2
\end{array}
$$

Assertion Language (extension)

Syntax (continued)

$$A \quad ::= \quad \ldots$$
$$\quad \mid \mathsf{emp}_{ep} \mid E \overset{peer}{\mapsto} (C\{a\}, E') \qquad \text{endpoint predicates}$$

Intuitively $E \overset{peer}{\mapsto} (C\{a\}, E')$ means :

- E is an allocated endpoint
- its peer is E'
- it is ruled by contract C
- it currently is in contract state a

Message Annotation

- We have to know the contents of messages
- Each message m appearing in a contract is described by a formula I_m of our logic.

- I_m may refer to two special variables:
  - `val` will denote the location of the message in memory
  - `src` will denote the location of the sending endpoint
- $I_m(x, f) \triangleq I_m[\mathtt{val}{\leftarrow}x, \mathtt{src}{\leftarrow}f]$

Proof System of Standard Separation Logic

Standard Hoare Logic

$$\frac{\{A\}\ p\ \{A'\} \qquad \{A'\}\ p'\ \{B\}}{\{A\}\ p; p'\ \{B\}} \qquad \ldots$$

Local Reasoning Rules

$$\frac{\{A\}\ p\ \{B\}}{\{A * F\}\ p\ \{B * F\}} \qquad \frac{\{A\}\ p\ \{B\} \qquad \{A'\}\ p'\ \{B'\}}{\{A * A'\}\ p || p'\ \{B * B'\}}$$

Small Axioms

$$\{A\}\ \mathsf{x} = \mathsf{E}\ \{A[x \leftarrow x'] \wedge x = E[x \leftarrow x']\}$$

$$\{\mathsf{emp}\}\ \mathsf{x} = \mathsf{new}()\ \{\exists v.\, x \mapsto v\} \qquad \ldots$$

Proof System (extended)

Standard Hoare Logic

Unchanged.

Local Reasoning Rules

Unchanged.

Small Axioms

Small axioms added for new commands.

Small Axioms for Communications

Open and Close rules:

$$\frac{i = \mathrm{init}(C)}{\{\mathsf{emp}\}\ (\mathsf{e},\mathsf{f}) = \mathsf{open}(\mathsf{C})\ \{e \overset{peer}{\mapsto} (C\{i\}, f) * f \overset{peer}{\mapsto} (\bar{C}\{i\}, e)\}}$$

$$\frac{a \in \mathrm{finals}(C)}{\{e \overset{peer}{\mapsto} (C\{a\}, f) * f \overset{peer}{\mapsto} (\bar{C}\{a\}, e)\}\ \mathsf{close}(\mathsf{e},\mathsf{f})\ \{\mathsf{emp}\}}$$

Small Axioms for Communications

Receive rule:

$$\frac{a \xrightarrow{?m} b \in C}{\{e \overset{peer}{\mapsto} (C\{a\}, f)\}\ \mathsf{x} = \mathsf{receive}(\mathsf{m},\ \mathsf{e})\ \{e \overset{peer}{\mapsto} (C\{b\}, f) * I_m(x, f)\}}$$

Small Axioms for Communications

Send rules:

$$\frac{a \xrightarrow{!m} b \in C}{\{e \overset{peer}{\mapsto} (C\{a\}, -) * I_m(E, e)\}\ \textsf{send}(\text{m,e,E})\ \{E \overset{peer}{\mapsto} (C\{b\}, -)\}}$$

$$\frac{a \xrightarrow{!m} b \in C}{\{e \overset{peer}{\mapsto} (C\{a\}, -) * (e \overset{peer}{\mapsto} (C\{b\}, -) \mathrel{-\!\!*} I_m(E, e))\}\ \textsf{send}(\text{m,e,E})\ \{\text{emp}\}}$$

Theorem 6 (Soundness for Copyless Message Passing)

If a Hoare triple $\{A\}$ p $\{B\}$ is provable and the contracts are leak free, then if the program p starts in a state satisfying A,

1. *contracts are respected*
2. *p does not fault on memory accesses*
3. *p does not leak memory*
4. *if p terminates, the final states satisfy B*
5. *there is no race*
6. *no communication error occur*

Theorem 6 (Soundness for Copyless Message Passing)

If a Hoare triple $\{A\}\ p\ \{B\}$ is provable and the contracts are leak free, then if the program p starts in a state satisfying A,

1. *contracts are respected*
2. *p does not fault on memory accesses*
3. *p does not leak memory* — *thanks to contracts!*
4. *if p terminates, the final states satisfy B*
5. *there is no race*
6. *no communication error occur* — *thanks to contracts!*

HEAP HOP

heaps that hop!

[TACAS'10] Tracking Heaps that Hop with Heap-Hop

`http://www.lsv.ens-cachan.fr/~villard/heaphop/`

Outline

In this Talk

- Formalization of heap-manipulating, message passing programs with contracts
- Contracts help us to ensure the absence of memory leaks
- Proof system
- Tool to prove specifications: Heap-Hop

In this Talk

- Formalization of heap-manipulating, message passing programs with contracts
- Contracts help us to ensure the absence of memory leaks
- Proof system
- Tool to prove specifications: Heap-Hop
- Not in this talk: Semantics, based on abstract separation logic

In this Talk

- Formalization of heap-manipulating, message passing programs with contracts
- Contracts help us to ensure the absence of memory leaks
- Proof system
- Tool to prove specifications: Heap-Hop
- Not in this talk: Semantics, based on abstract separation logic

In a Future Talk

- Contracts help us to ensure the absence of deadlocks
- Enrich contracts with counters, non determinism, . . .
- Tackle "real" case studies: MPI, cache coherence protocols, . . .