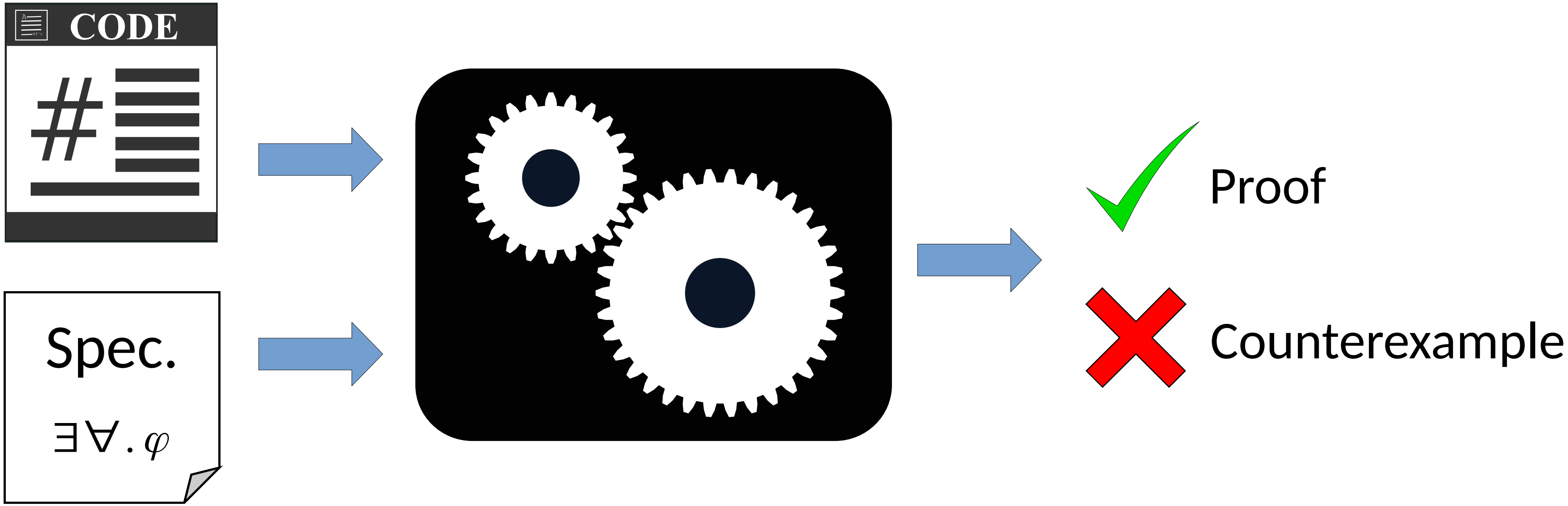
Verifying Message-Passing Systems

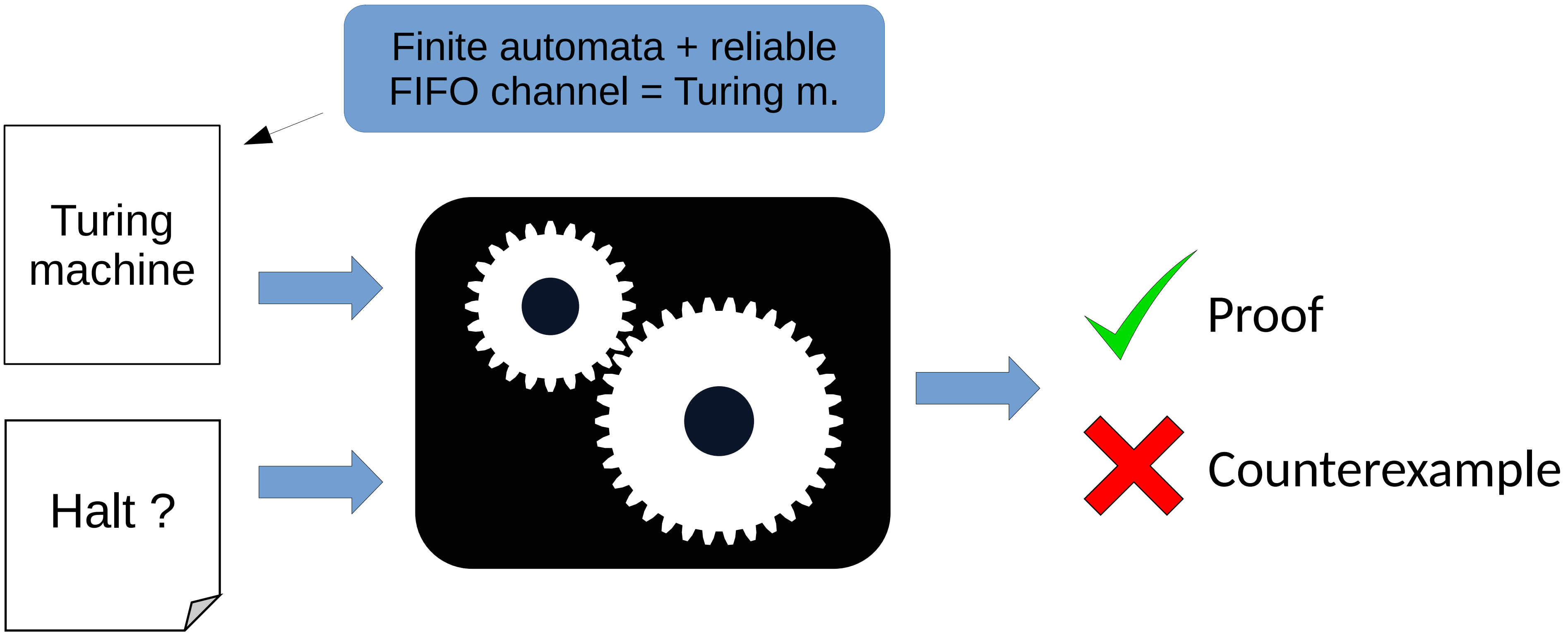
Damien Zufferey MPI-SWS

Dagstuhl 2019
28.10.2019

Automated Software Verification



Unfortunately ...



Problems Do Not Go Away Because They are Hard...

… but they still make life miserable.

Maybe the problem is ill-posed:

- Full automation is too much (ask the user for help)
- Try an easier problem (focus on null pointers)
- Programming languages are too powerful
- ...

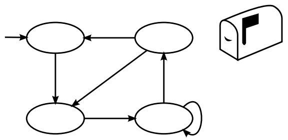
Using PL to Make Verification Easier

Automated verification requires breaking a problem in small enough parts so the verification becomes “simple.”

Try to build a (de)composition strategy that the verification can use into the programming abstraction.

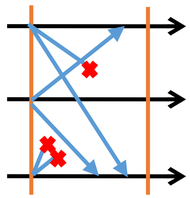
My research focus on doing that for message passing programs.

Some Earlier Work



P(rotocol) [PLDI 13]

- Communicating state-machines
- Decomposition: control and data
- Applied to device drivers in Windows

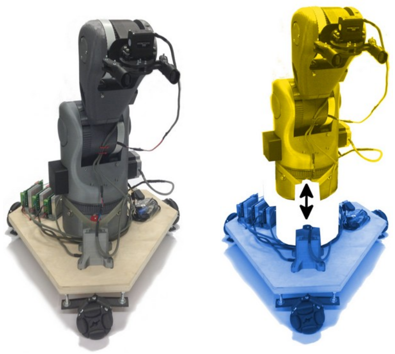


PSync (partial synchrony) [POPL 2016]

- Communication-closed rounds
- Decomposition: “time”
- Applied to fault-tolerant distributed algorithms

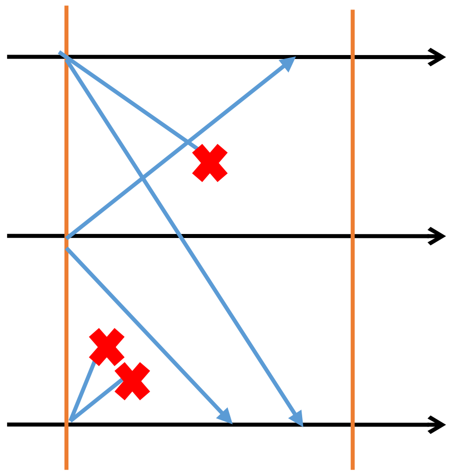
More Recent: Cyber-Physical Systems (CPS)

CPS = Communication + Computation + Control



- PGCD (Geometry, Concurrency, and Dynamics) [ICCPS 19, ECOOP 19]
  - Decomposition: communication and local perspective
  - Application: modular robotics

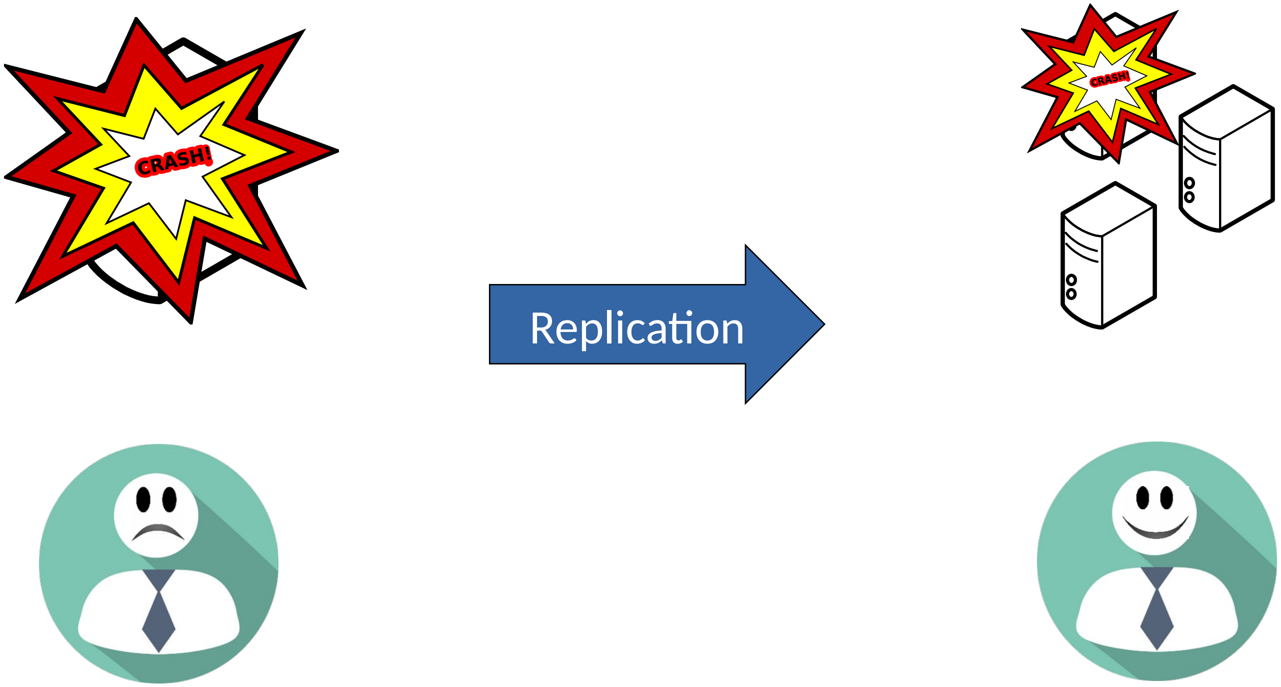
PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms



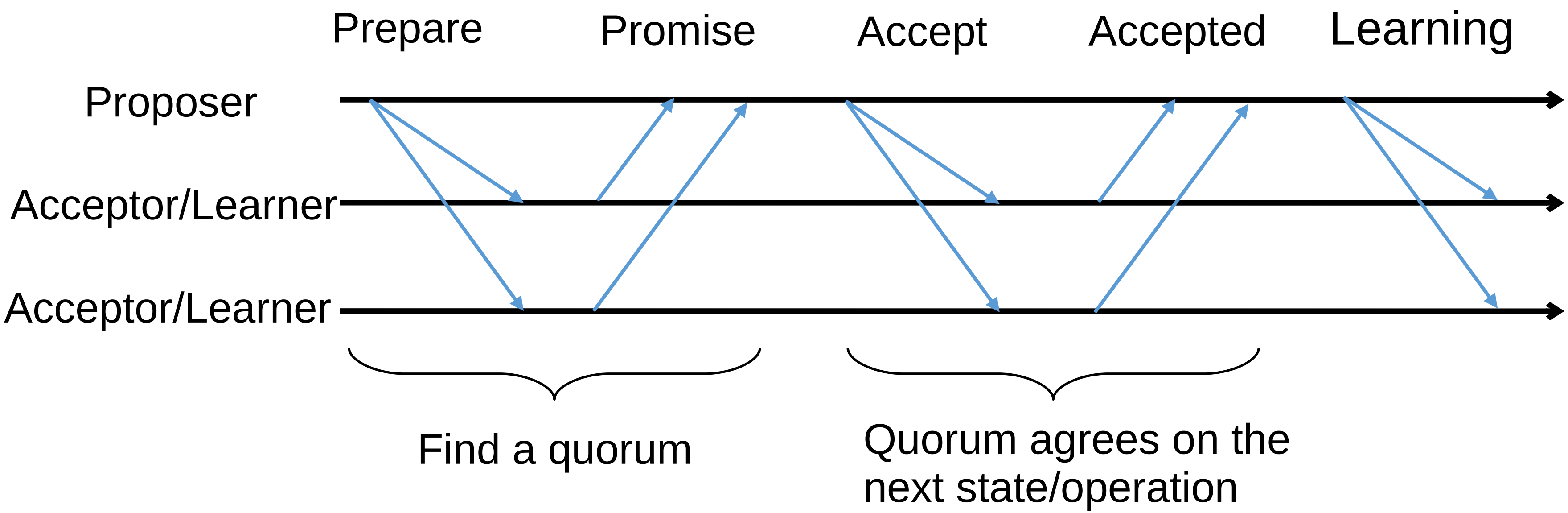
Collaboration with Cezara Drăgoi, Thomas A. Henzinger, Josef Widder, Helmut Veith

[POPL 16, SNAPL 15, VMCAI 14] and ongoing work

Why Care About Failure?



The Paxos Algorithm [Lamport 98]

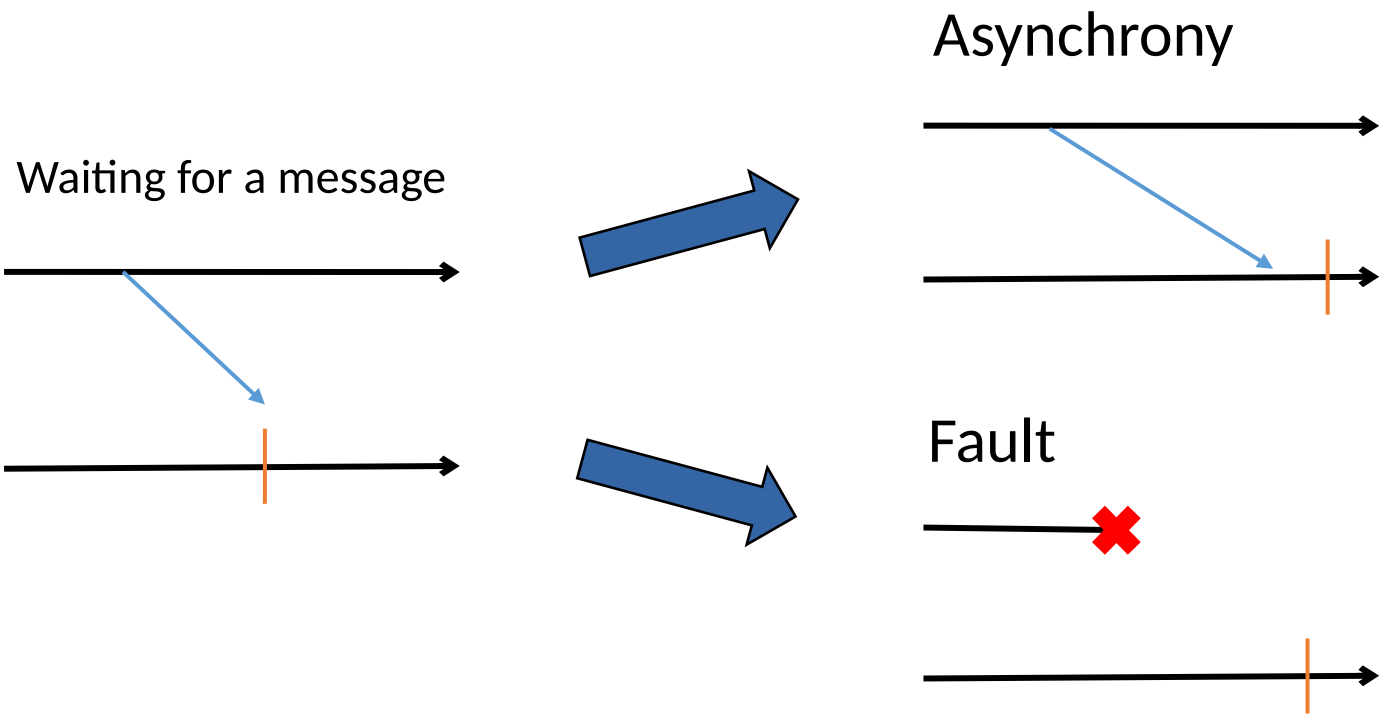


Implementing Paxos: from ~50 lines of pseudo code to >500 LoC. What goes wrong?

Implementation Challenges

- Detecting failure (and the impossibility to get it right)
- Messages (side-effects) are untyped, have no scope, etc.
- Control-flow inversion (losing the program structure)

When Processes Fail?



Wait longer can help but how to avoid blocking?

Asynchronous models are not good at dealing with faults.

[FLP 85] Solving consensus requires time.

Communication is a Side Effect

```
Type object;
Byte[] buffer =
        serialize(obj);
send(channel, buffer);
```

011001010101...

```
buffer = recv(channel);
Type1 object1 =
    deserialize1(buffer);
…
buffer = recv(channel);
Type2 object2 =
    deserialize2(buffer);
```

Up to the programmer to:
- interpret the bytes moving over the network,
- know which receive corresponds to which send.

Control-flow Inversion

Protocol structure replaced by dispatch:

Protocol:
(1)Msg A
(2)Msg B

```
var state = 1

while (true) {
    on receive {
        case Msg A =>
            if (state == 1) …
            else if (state == 2) …
        case Msg B =>
            if (state == 1) …
            else if (state == 2) …
    }
}
```

normal case

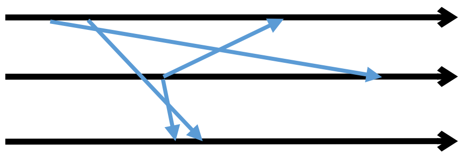
message duplicated

message dropped

normal case

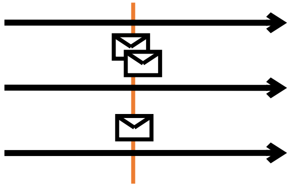
Verification Perspective

- Asynchrony



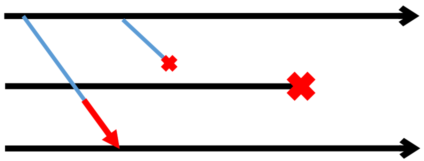
Interleavings → combinatorial explosion

- Channels



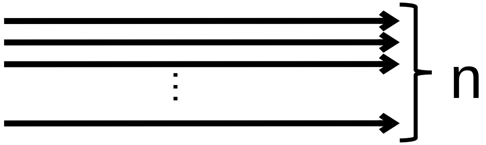
reliable FIFO → undecidable

- Faults



another combinatorial explosion

- Parametric systems

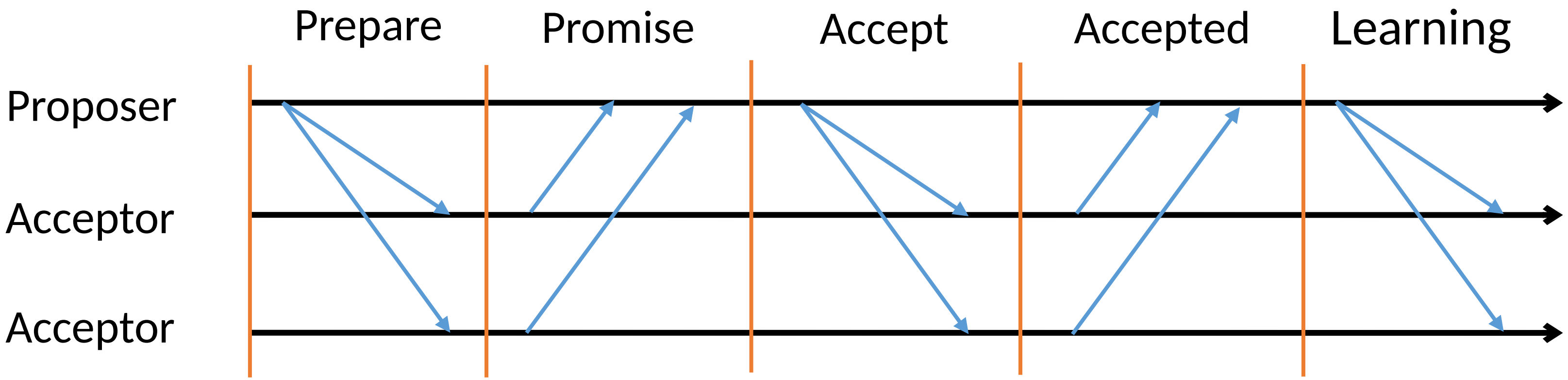


usually undecidable

PSync

- Goals: **make things simpler** (to understand and verify)
  - The code preserves the algorithm structure
  - Code complexity on par with pseudo-code
- How: **communication-closed rounds**
  - Syntactic scope to the messages
  - Provides logical time
  - Gives the *illusion* of synchrony

Communication-closed Rounds
[Elrad & Francez 82]



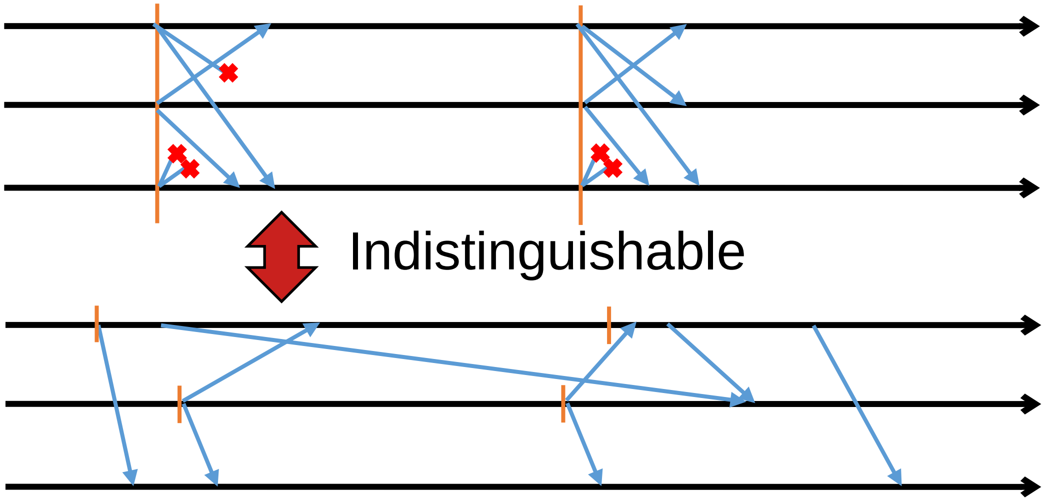
Paxos organized in communication-closed rounds.

No message cross the boundaries between rounds.

How Does It Model a Real System?

Idea: model faults/asynchrony as an adversarial environment [Gafni 98]
Project all the "faults" on the messages

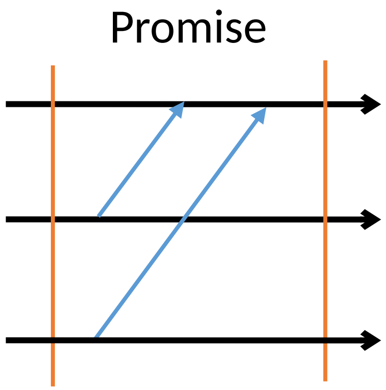
Lockstep semantics:



Asynchronous execution:

Local views are preserved.

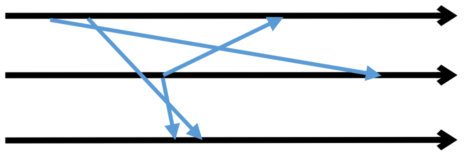
Round Example



```
new Round[(Int,Time)]{

    def send(): Map[ProcessID, (Int,Time)] =
        Map(coord -> (x, ts))

    def update(mailbox: Map[ProcessID, (Int,Time)]) {
        if (id == coord && mailbox.size > n/2) {
            vote = valueWithMaxTS(mailbox)
            commit = true
        }
    }
}
```

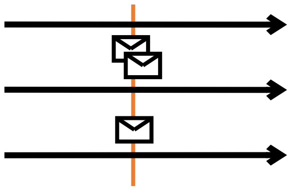
Benefits for the Verification

- Asynchrony



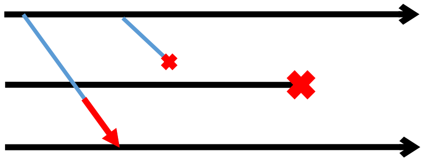
No interleaving

- Channels



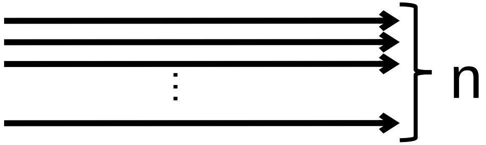
Buffer of size 1

- Faults



Project all faults on messages
(less nondeterminism)

- Parametric systems



Decidable with further restrictions
[Marić et al. CAV 17]

Problem: Performance and User-Control

Old version, runtime is a blackbox. No control:

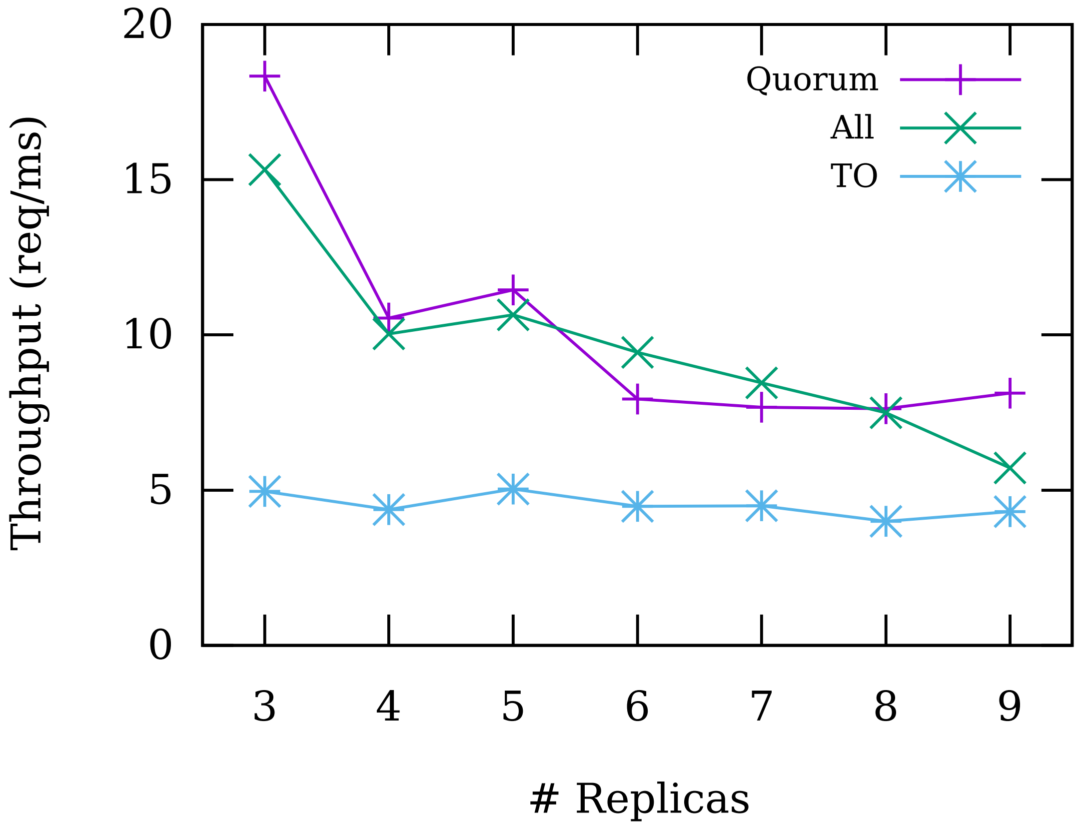
```
new Round[(Int,Time)]{

    def send(): Map[ProcessID, (Int,Time)] =
        Map(coord -> (x, ts))

    def update(mailbox: Map[ProcessID, (Int,Time)]) {
        if (id == coord && mailbox.size > n/2) {
            ...
        }
    }
}
```

Problem: Performance and User-Control

New version, the program can give hints to the runtime:

```
new Round[(Int,Time)]{
    var nMsg = 0
    def init =
        if (id == coord) Progress.timeout( timeout )
        else             Progress.goAhead
    def send(): Map[ID, (Int,Time)] =
        Map(coord -> (x, ts))
    def receive(sender: ID, payload: (Int, Time)) = {
        nMsg += 1
        ...
        if (nMsg > n/2) Progress.goAhead
        else            Progress.unchanged
}   }
```

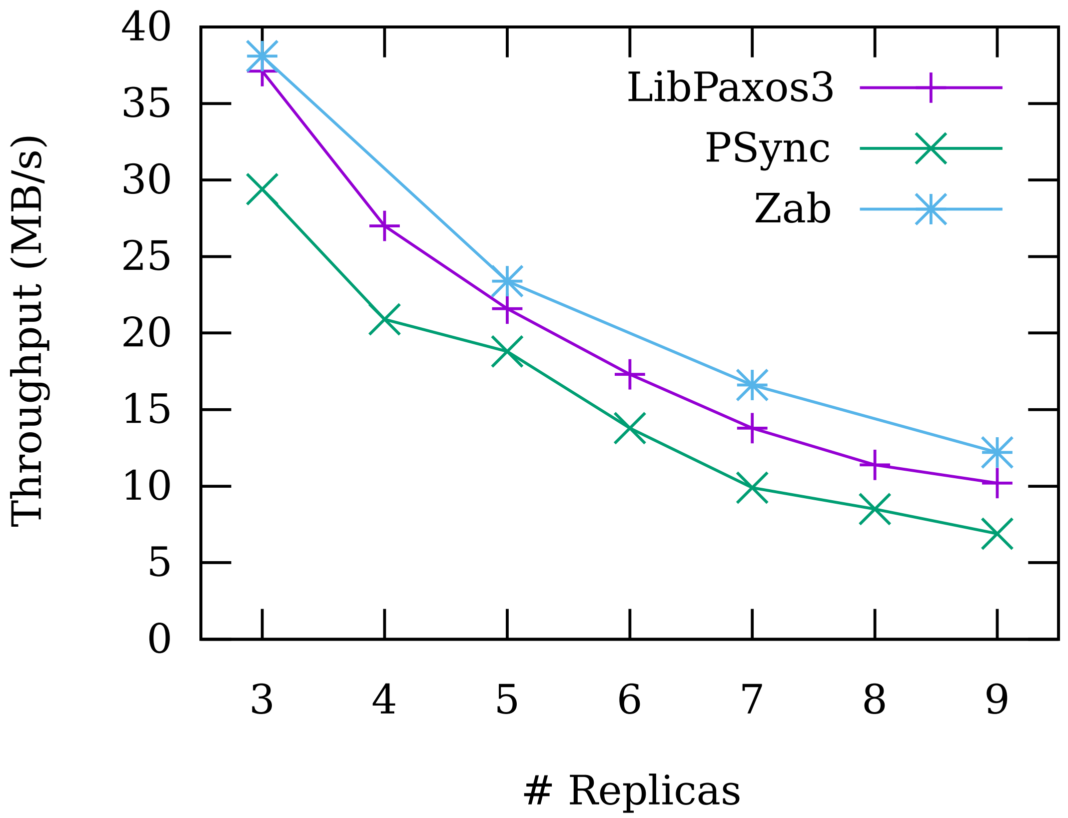
Results



Same algorithm but with different progress conditions.

Timeout (TO) is the best the old version can do will providing liveness guarantees.

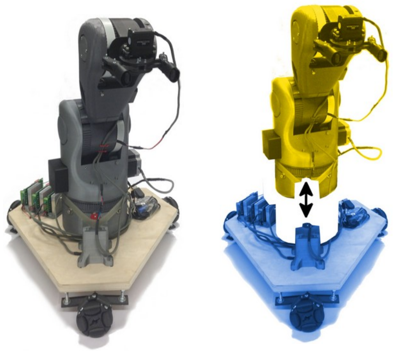
Results



LibPaxos3
PSync
Zab
Throughput (MB/s)
Replicas
0
5
10
15
20
25
30
35
40
3
4
5
6
7
8
9

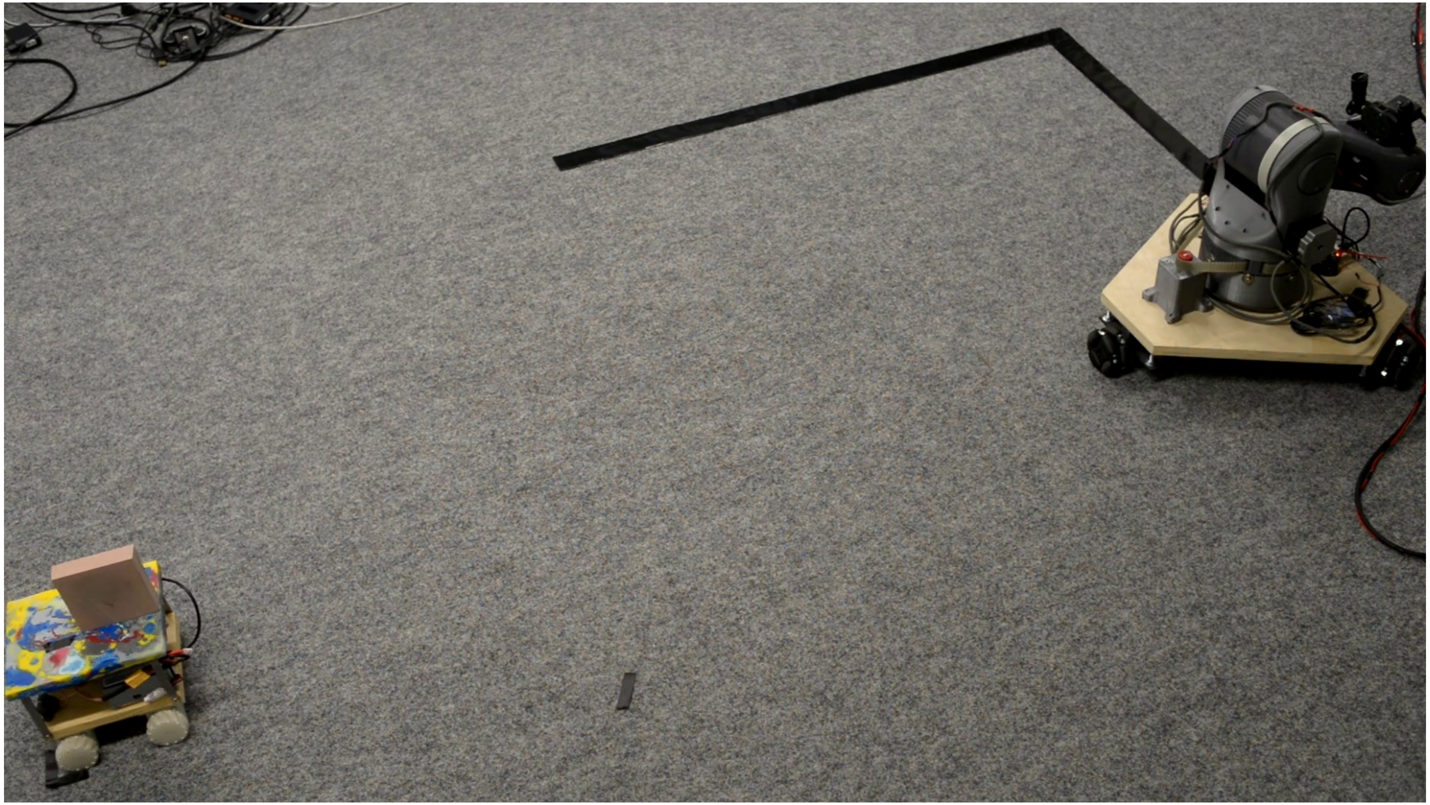
Current Work: Cyber-Physical Systems (CPS)

CPS = Communication + Computation + Control

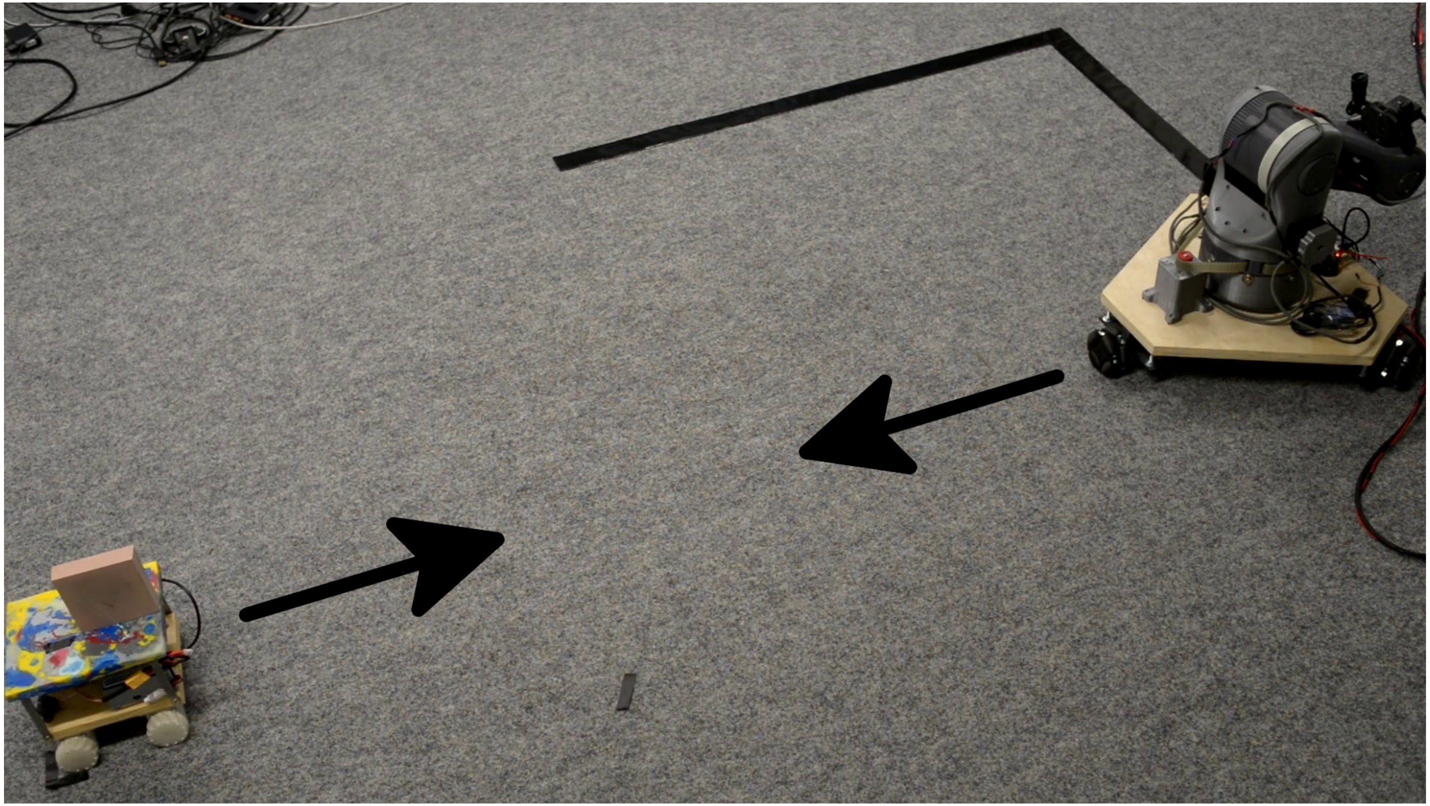


- PGCD (Geometry, Concurrency, and Dynamics) [ICCPS 19, ECOOP 19]
  - Decomposition: communication and local perspective
  - Application: modular robotics

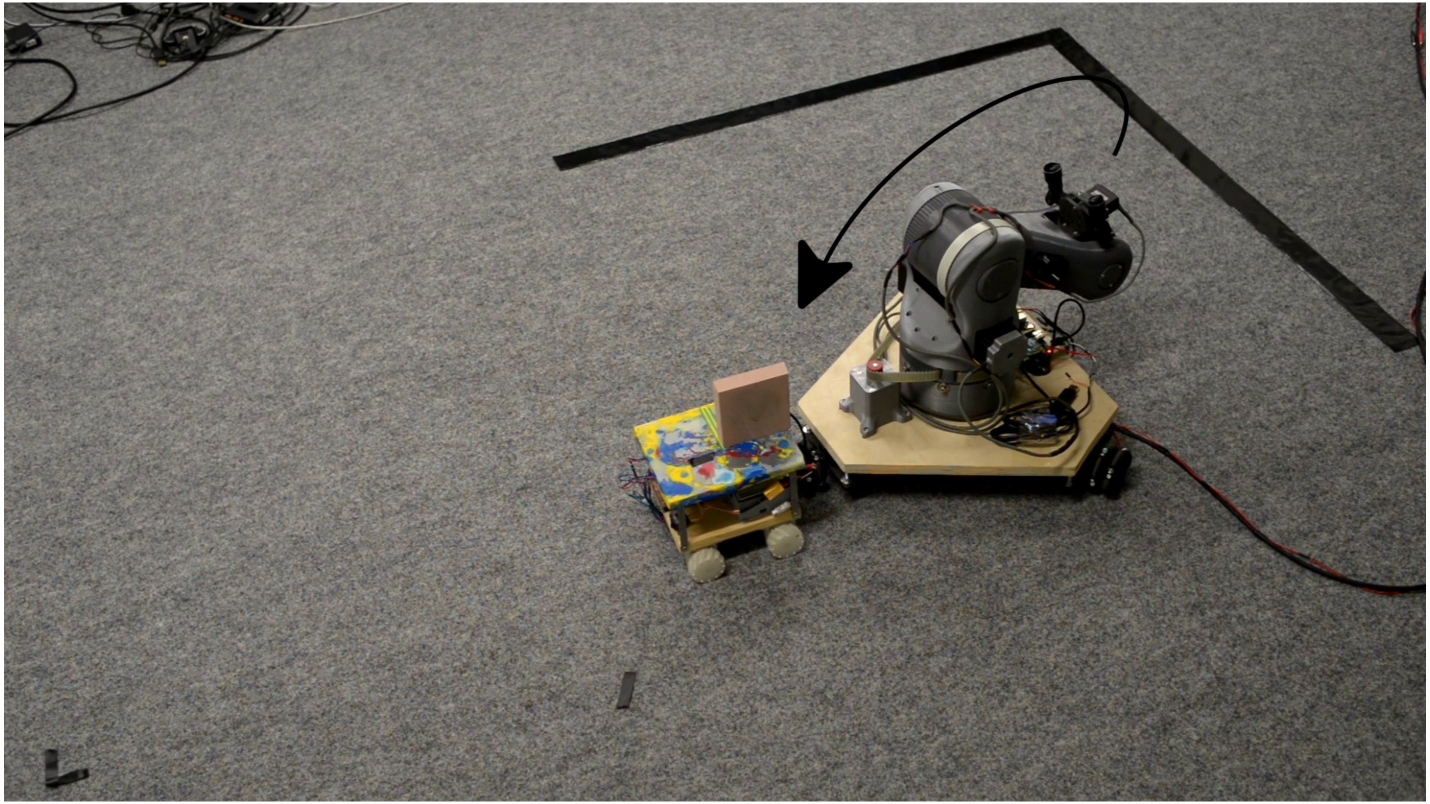
Example: Handover



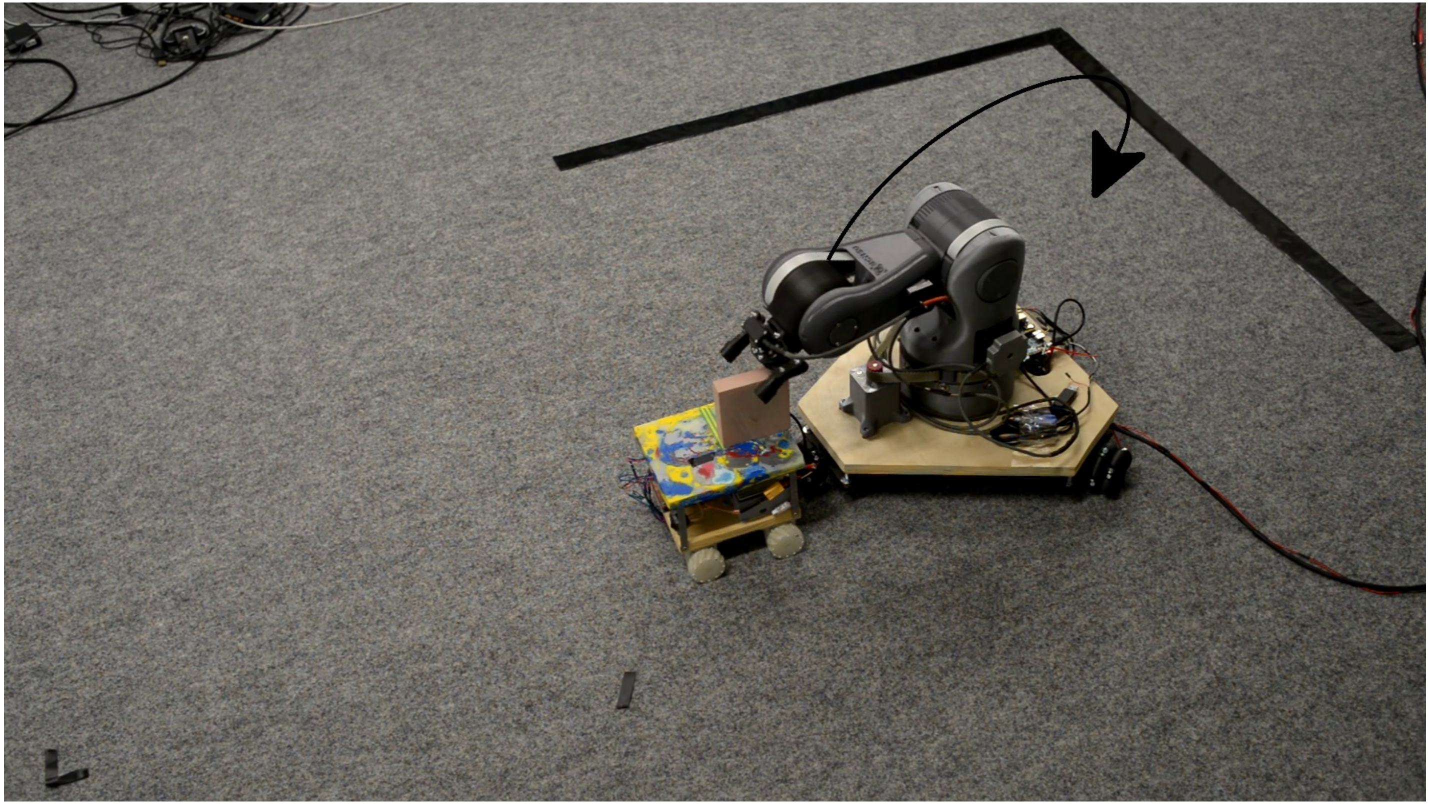
Example: Handover (Meet)



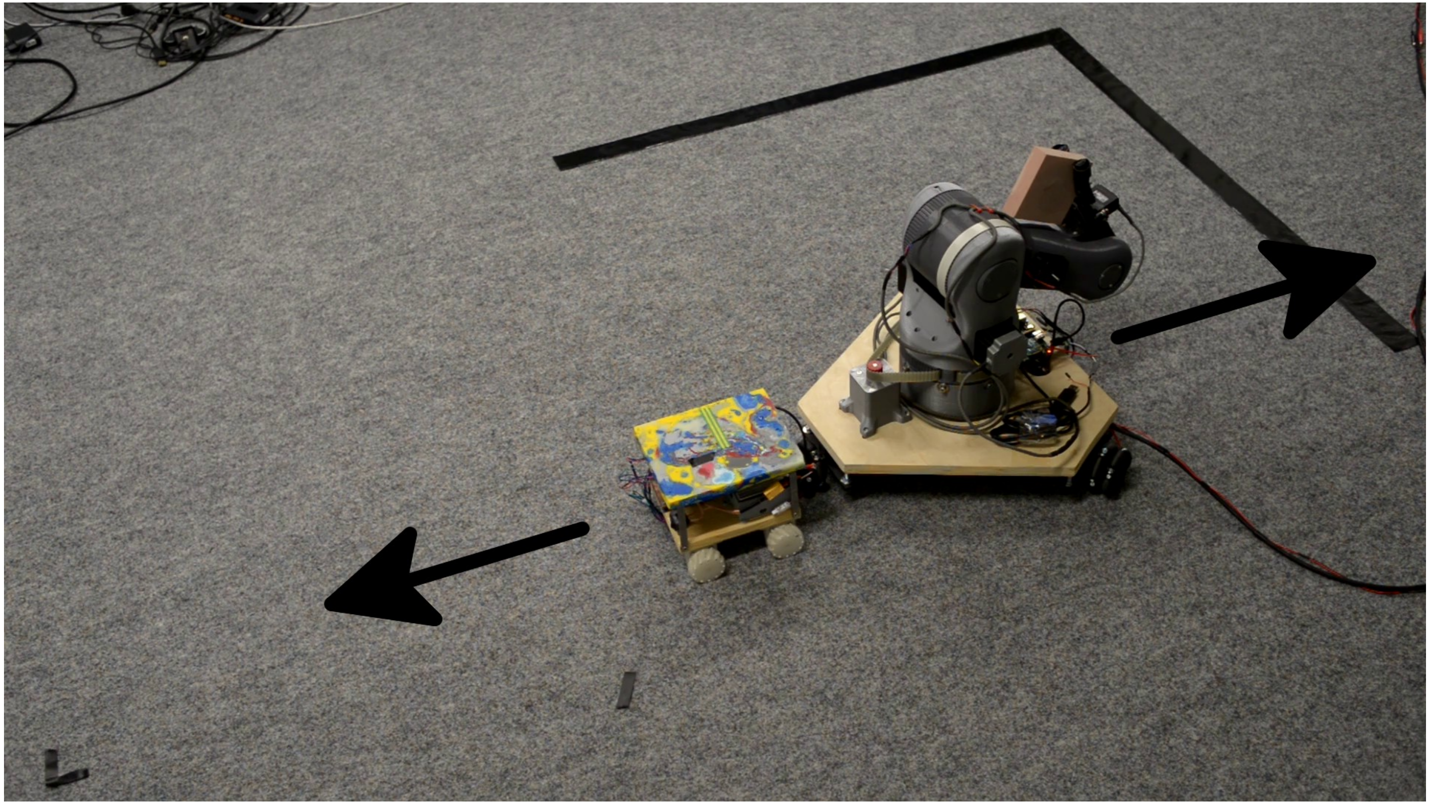
Example: Handover (Grab)



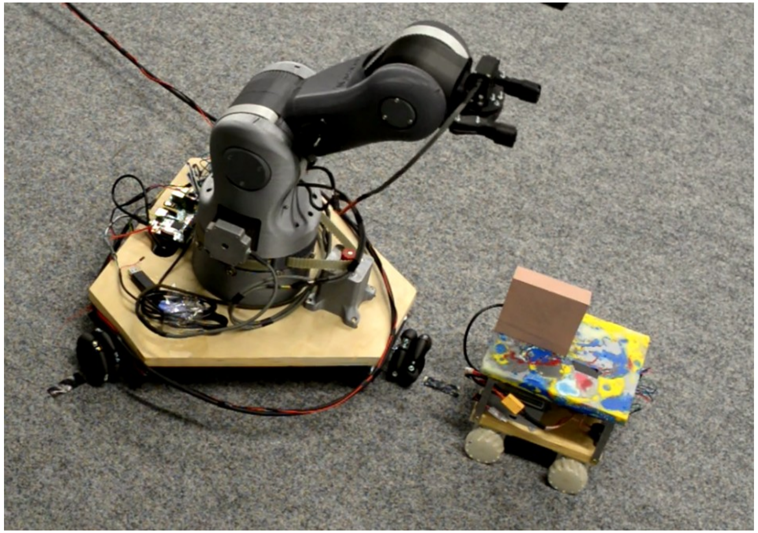
Example: Handover (Fold)



Example: Handover (Done)



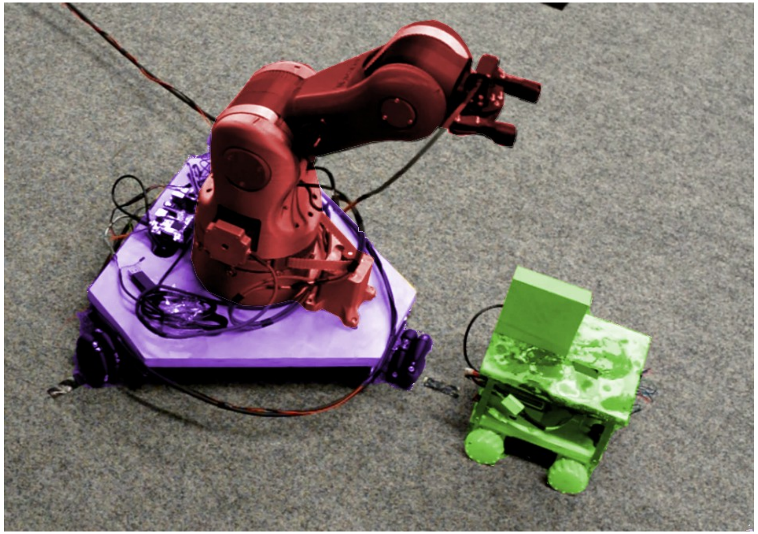
How Many Robots?



How Many Robots?

Arm

Cart

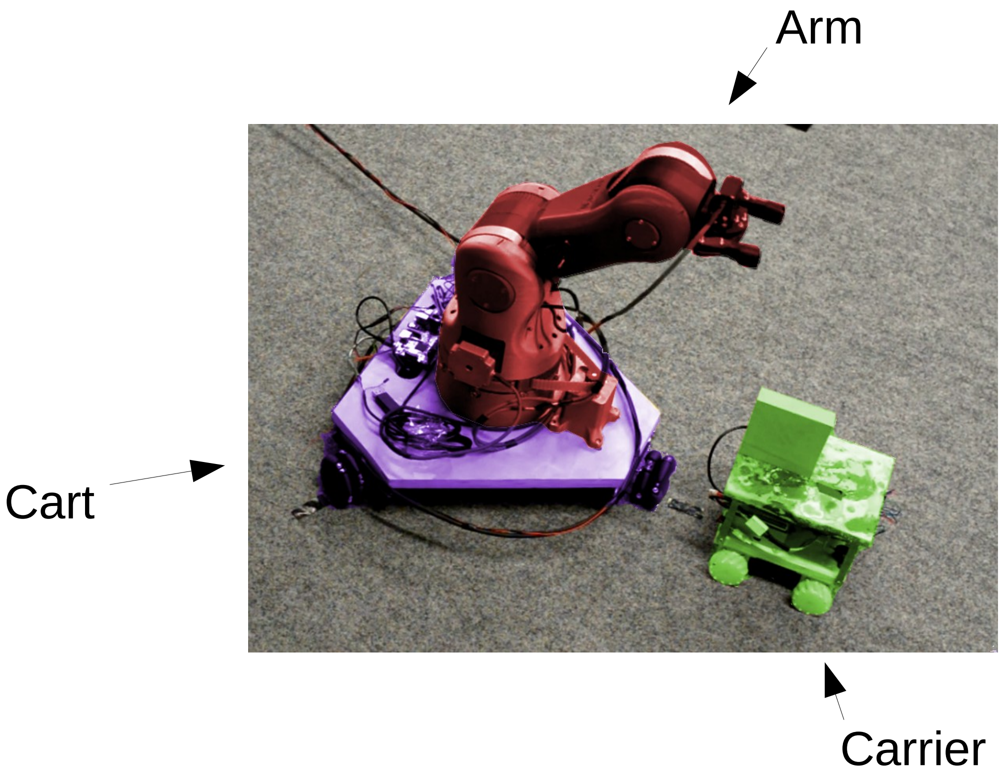


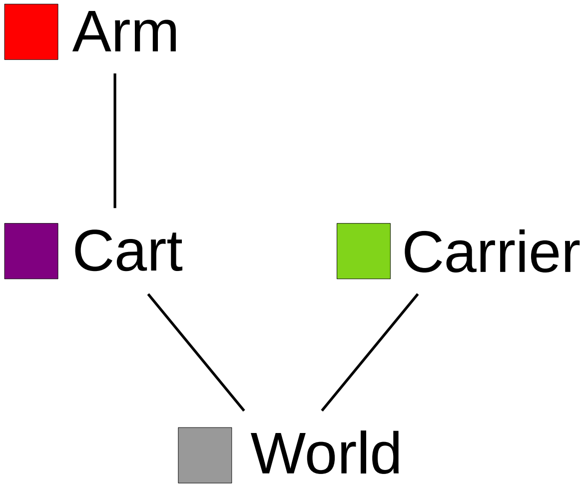
Carrier

Cart & Arm:

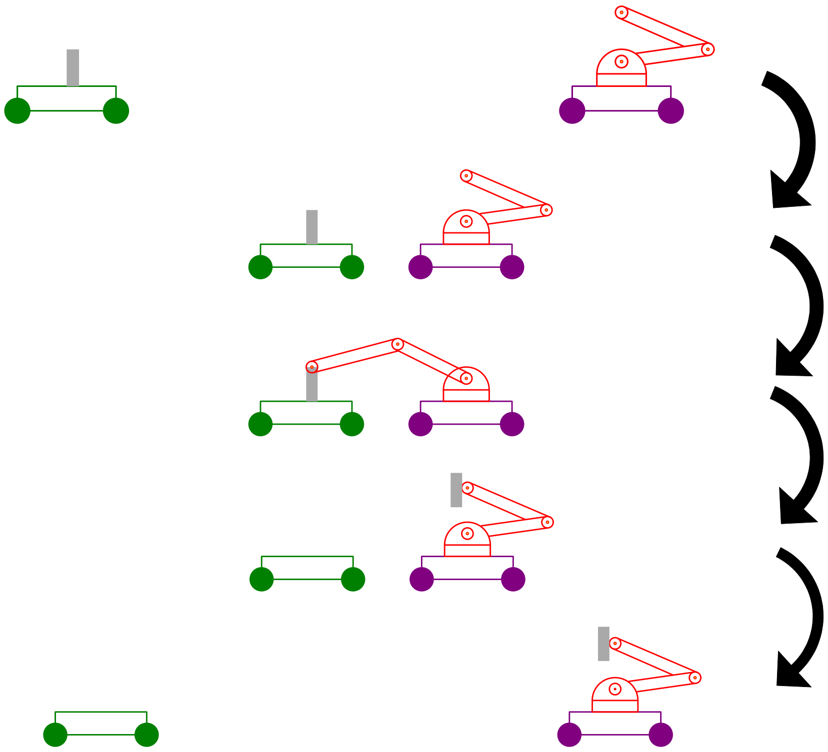
Two robots attached together that act as “one” robot (communication)

How Many Robots?

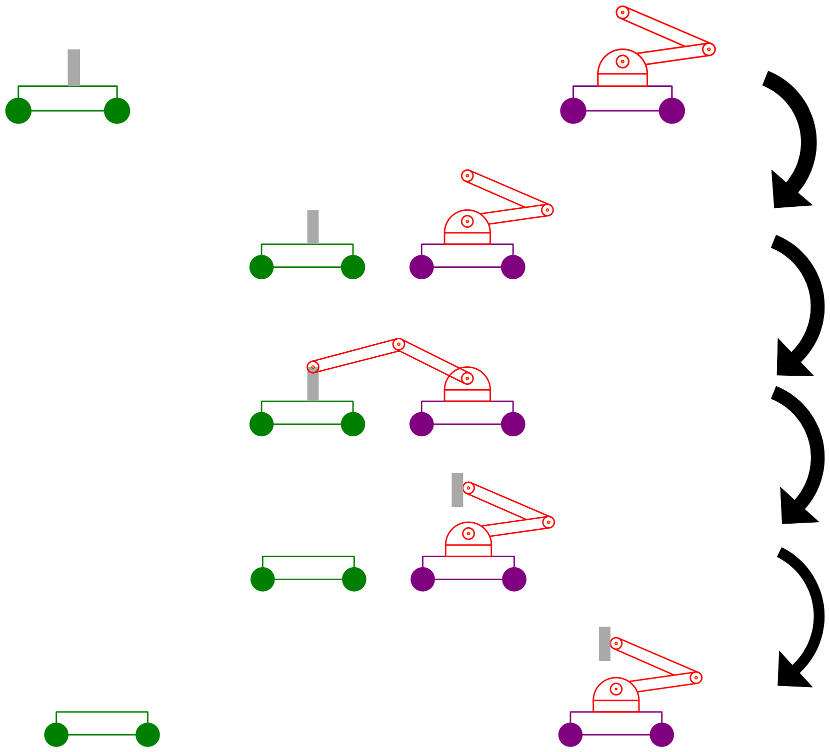




Sequence of Actions



Arm's Code



```
while true do
  receive (Cart, idle)
    grab(loc) ⇒
      grip(loc);
      send(cart, ok)
    fold ⇒
      move(origin);
      send(cart, ok)
    done ⇒
      break
```

Robotic Program(mer)s as Target

- Message-Passing Abstraction
  - Robotic Operating System (ROS): publish-subscribe model
  - ROS is a de facto standard in academia (used for prototyping in industry)
- Modular robots
  - Software: logically separate units
  - Hardware: in a shared world (physical coupling)
- Different “kinds” of code
  - Planning: synchronize with other robots, decide what to do next, ...
  - Control: following trajectories, actuation, sensing, ...

Modularity for Components

Cart || Arm

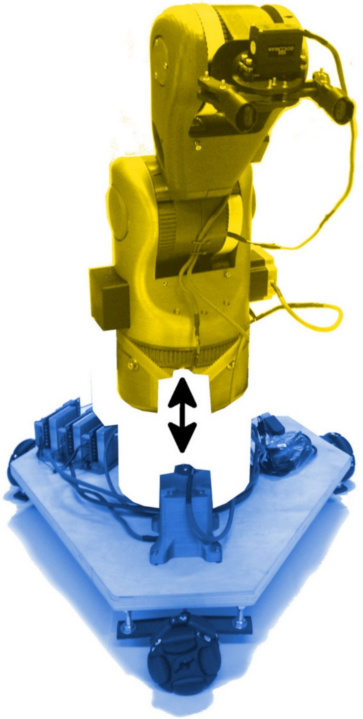
Traditional parallel composition
lacks information.

$\text{Cart} \ \triangleleft_{\theta, M} \ \text{Arm}$

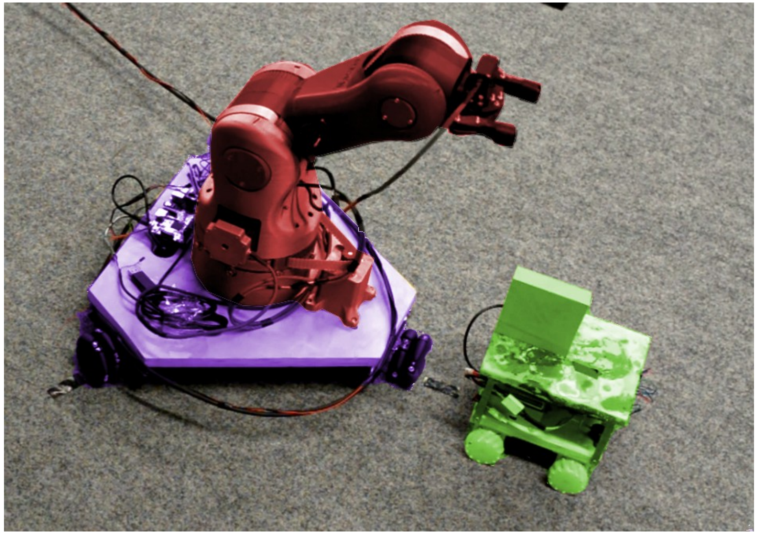
Oriented operator:
- cart is the parent
- arm is the child

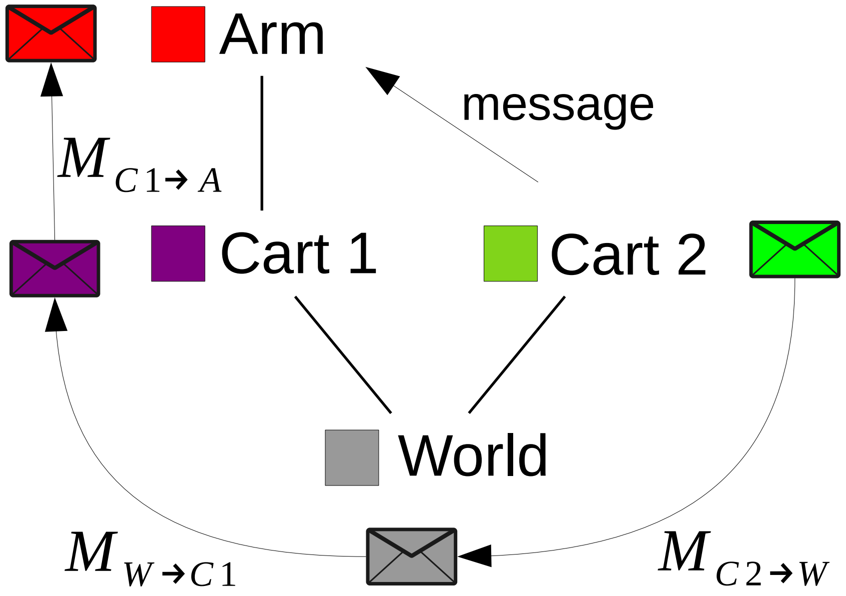
Dynamic coupling:
Cart’s speed depends
on arm’s weight.

Frame shift (relative position):
Arm’s position depends on Cart

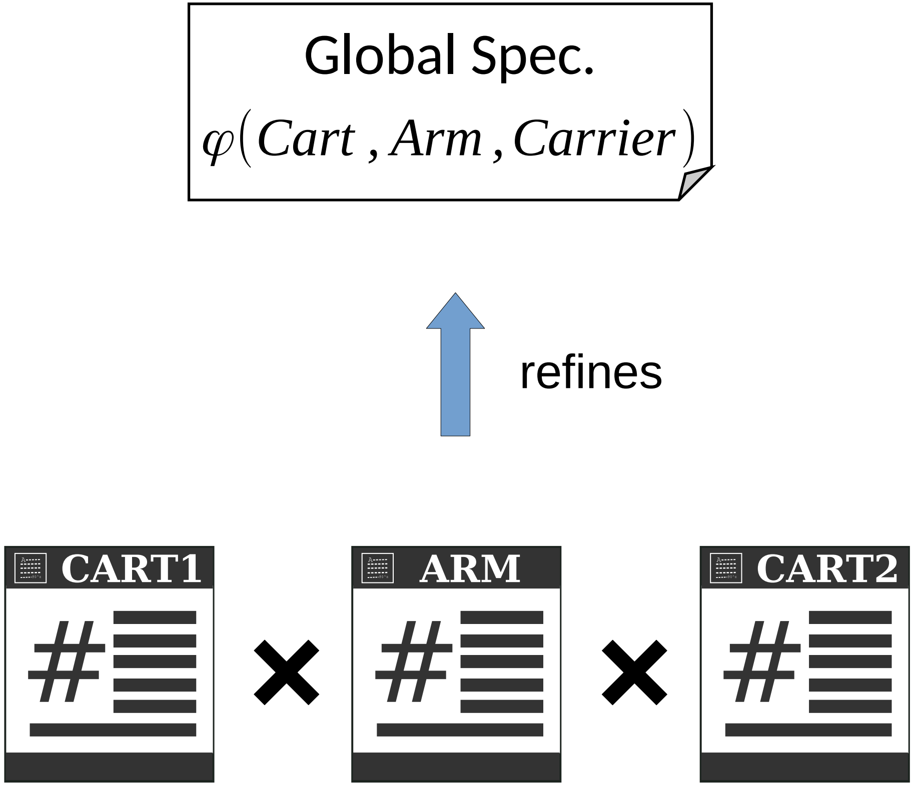


Making Composition Transparent





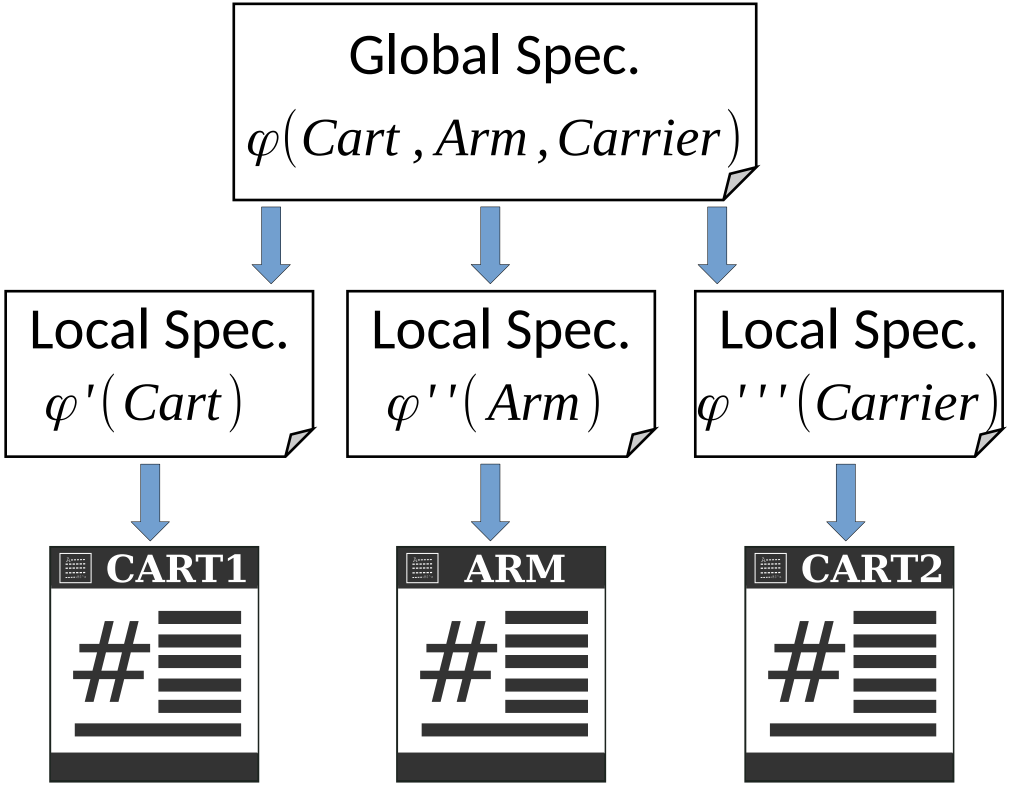
Goal: Verification



The product is expensive.
(state-space explosion!)

Assume-Guarantee helps
for bottom-up composition.

Decomposing with Sessions Typs



Motion session type (Global)

Projections

Motion session type (Local)

Typing

Implementation

Global Types

$$
\begin{array}{lll}
G & ::= & \mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle.G \\
 & \mid & \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i \in I} \\
 & \mid & \mu \mathbf{t}.G \quad \mid \quad \mathbf{t} \\
 & \mid & \mathtt{end}
\end{array}
$$

Motion for each process

Communication & branching
(same sender+receiver for all choices)

Recursion
(no argument → regular language)

Global descriptions of the messages and the motions happening in the robots.

Local Types

$$
\begin{array}{rcll}
T & ::= & \mathsf{dt}\langle a \rangle.T & \\
& \mid & \&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I} & \\
& \mid & \&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I} \;\&\; \mathsf{dt}\langle a \rangle.T & \\
& \mid & \oplus\{\mathsf{q}!\ell_i(S_i).T_i\}_{i\in I} & \\
& \mid & \mu\mathbf{t}.T \quad \mid \quad \mathbf{t} & \\
& \mid & \mathtt{end} &
\end{array}
$$

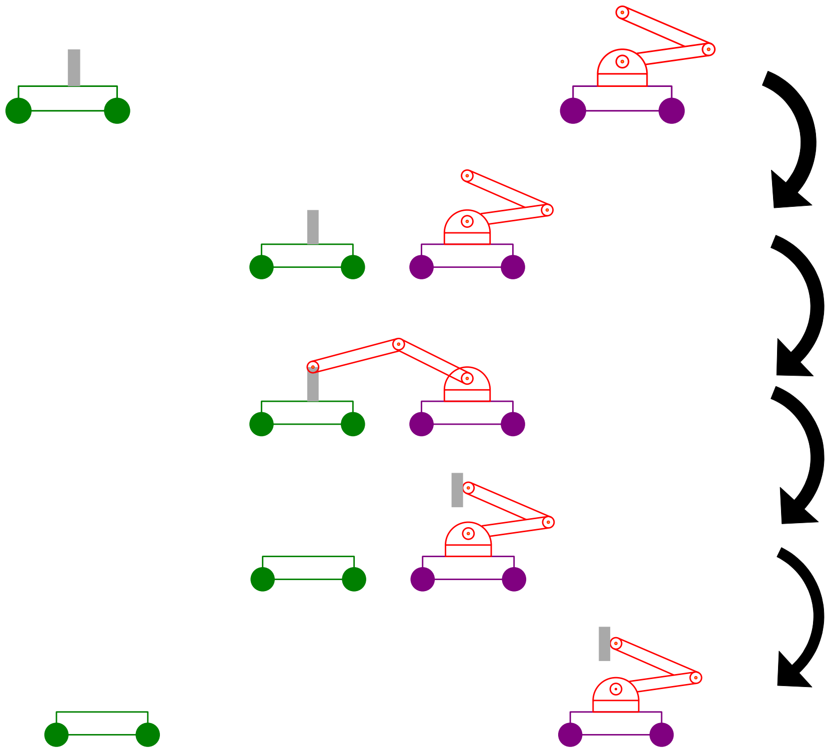
Motion

External choice
(receive + optional motion)

Internal choice + send

No mixed (internal+external) choice.
No send/receive with more than one other process.
Time is like a message broadcast to every process.

Handover: Global Type



$\mathsf{Cart} \to \mathsf{Arm} : \mathit{fold}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Carrier} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold}\rangle.$

$\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{ok}(\mathtt{unit}).$

$\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Carrier} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.$

$\mathsf{Carrier} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Arm} : \mathit{grab}(\mathtt{unit}).$

$\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Carrier} : \mathsf{idle}, \mathsf{Arm} : \mathsf{grip}\rangle.$

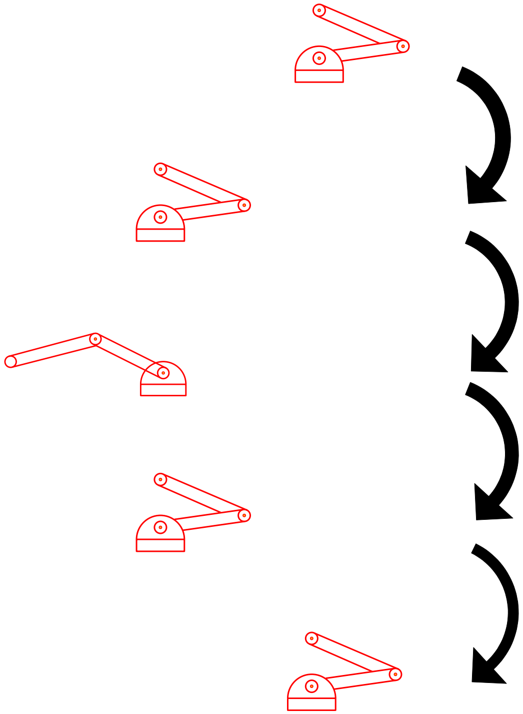
$\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{ok}(\mathtt{unit}).$

$\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Carrier} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.$

$\mathsf{Cart} \to \mathsf{Arm} : \mathit{done}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{done}(\mathtt{unit}).\mathtt{end}$

■	Motion primitive
■	Communication

Handover: Projection on the Arm



$\mathsf{Cart} \to \mathsf{Arm} : \mathit{fold}(\mathtt{unit}).\mathsf{dt}\langle \qquad \mathsf{Arm} : \mathsf{fold}\rangle.$

$\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).$

$\mathsf{dt}\langle \qquad \mathsf{Arm} : \mathsf{idle}\rangle.$

$\mathsf{Cart} \to \mathsf{Arm} : \mathit{grab}(\mathtt{unit}).$

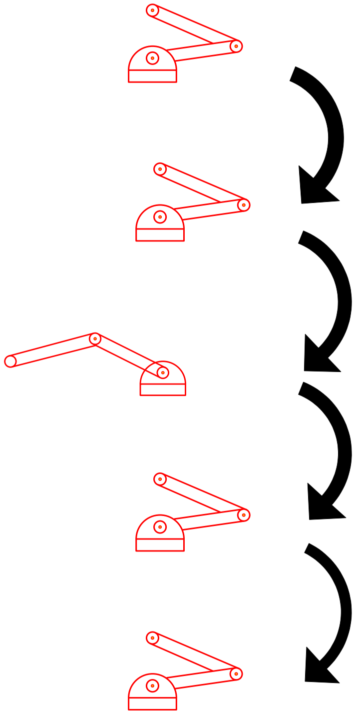
$\mathsf{dt}\langle \qquad \mathsf{Arm} : \mathsf{grip}\rangle.$

$\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).$

$\mathsf{dt}\langle \qquad \mathsf{Arm} : \mathsf{idle}\rangle.$

$\mathsf{Cart} \to \mathsf{Arm} : \mathit{done}(\mathtt{unit}). \qquad \mathtt{end}$

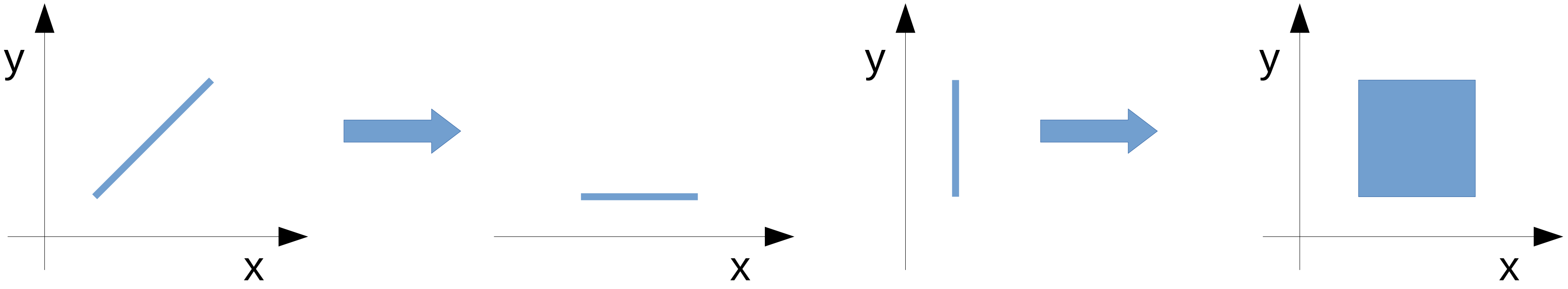
Handover: Projection on the Arm



$$
\begin{aligned}
&\mathsf{Cart}?\mathit{fold}(\mathtt{unit}).\\
&\qquad\qquad \mathsf{dt}\langle \mathsf{fold}\rangle.\\
&\qquad\qquad \mathsf{Cart}!\,\mathit{ok}(\mathtt{unit}).\\
&\mathsf{dt}\langle \mathsf{idle}\rangle.\\
&\mathsf{Cart}\,?\,\mathit{grab}(\mathtt{unit}).\\
&\qquad\qquad \mathsf{dt}\langle \mathsf{grip}\rangle.\\
&\qquad\qquad \mathsf{Cart}!\,\mathit{ok}(\mathtt{unit}).\\
&\mathsf{dt}\langle \mathsf{idle}\rangle.\\
&\mathsf{Cart}?\mathit{done}(\mathtt{unit}).\mathtt{end}
\end{aligned}
$$

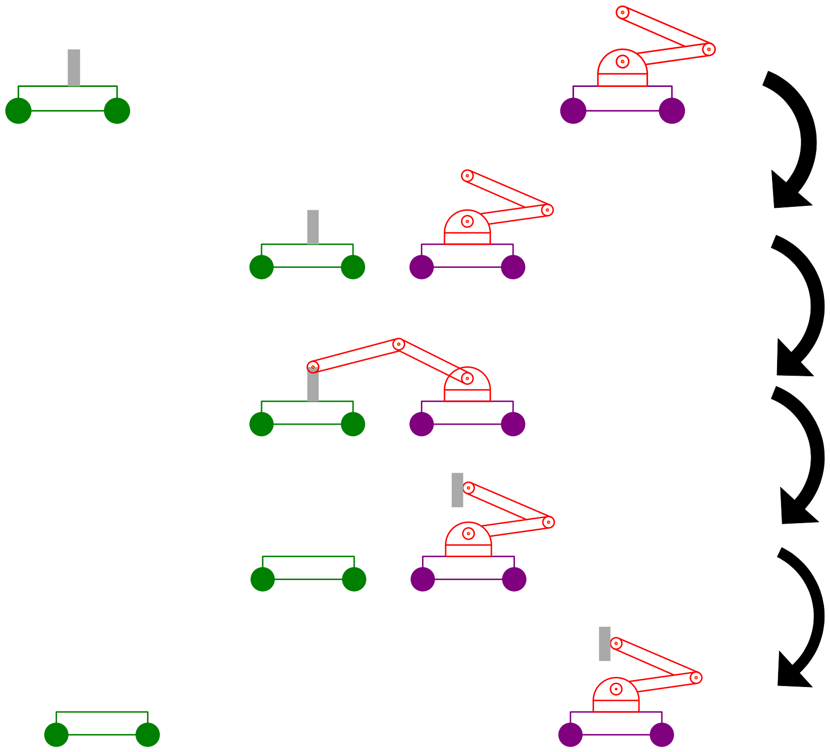
Projection is Tricky

- "Projection then product" usually adds behaviors. (geometric analogy)



- Multiparty Session Type has a behavior preserving projection.
  - Only works on a *restricted* class of protocols.

Arm’s Code



```
while true do
  receive (Cart, idle)
    grab(loc) ⇒
      grip(loc);
      send(cart, ok)
    fold ⇒
      move(origin);
      send(cart, ok)
    done ⇒
      break
```

Typing the Arm: Not So Straightforward

```
while true do
    receive (Cart, idle)
        grab(loc) ⇒
            grip(loc);
            send(cart, ok)
        fold ⇒
            move(origin);
            send(cart, ok)
        done ⇒
            break
```

$\mathsf{Cart}?\mathit{fold}(\mathtt{unit}).$

$\quad \mathsf{dt}\langle \mathsf{fold}\rangle.$

$\quad \mathsf{Cart}!\mathit{ok}(\mathtt{unit}).$

$\mathsf{dt}\langle \mathsf{idle}\rangle.$

$\mathsf{Cart}?\mathit{grab}(\mathtt{unit}).$

$\quad \mathsf{dt}\langle \mathsf{grip}\rangle.$

$\quad \mathsf{Cart}!\mathit{ok}(\mathtt{unit}).$

$\mathsf{dt}\langle \mathsf{idle}\rangle.$

$\mathsf{Cart}?\mathit{done}(\mathtt{unit}).\mathtt{end}$

Subtyping: Receiving Messages

Common parts preserve subtyping.

$$[\textsc{sub-in2}]\quad \frac{\boxed{\forall i \in I : \quad S_i' \leq: S_i \qquad T_i \leqslant T_i'}}{\&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in \boxed{I} \cup \boxed{J}} \ \& \ \boxed{\mathsf{dt}\langle a\rangle.T} \quad \leqslant \quad \&\{\mathsf{p}?\ell_i(S_i').T_i'\}_{i\in \boxed{I}}}$$

Can have more messages and motions.
(Never exercised)

Using Subtyping

Loop unfolded 3 times:

```
receive (Cart, idle)
      grab(loc) ⇒
            grip(loc);
            send(cart, ok)
      fold ⇒
            move(origin);
            send(cart, ok)
      done ⇒
            break
receive (Cart, idle)
      grab(loc) ⇒
            grip(loc);
            send(cart, ok)
      fold ⇒
            move(origin);
            send(cart, ok)
      done ⇒
            break
receive (Cart, idle)
      grab(loc) ⇒
            grip(loc);
            send(cart, ok)
      fold ⇒
            move(origin);
            send(cart, ok)
      done ⇒
            break
```

$\mathsf{Cart}?\mathit{fold}(\mathtt{unit}).$

$\quad \mathsf{dt}\langle \mathsf{fold} \rangle.$

$\quad \mathsf{Cart}!\mathit{ok}(\mathtt{unit}).$

$\mathsf{dt}\langle \mathsf{idle} \rangle.$

$\mathsf{Cart}?\mathit{grab}(\mathtt{unit}).$

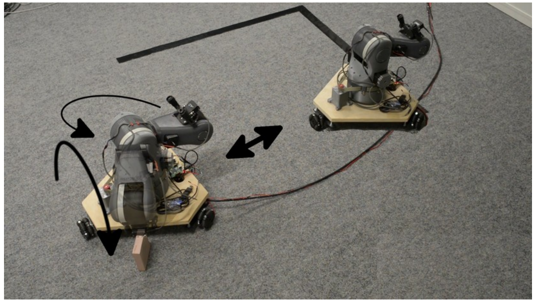
$\quad \mathsf{dt}\langle \mathsf{grip} \rangle.$

$\quad \mathsf{Cart}!\mathit{ok}(\mathtt{unit}).$

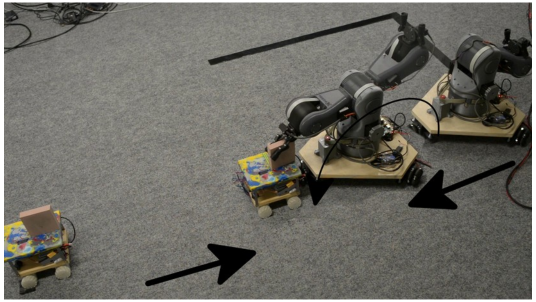
$\mathsf{dt}\langle \mathsf{idle} \rangle.$

$\mathsf{Cart}?\mathit{done}(\mathtt{unit}).\mathsf{end}$

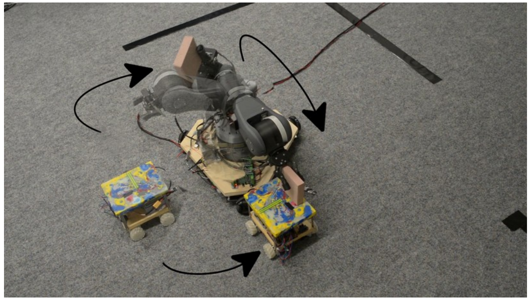
Experiments



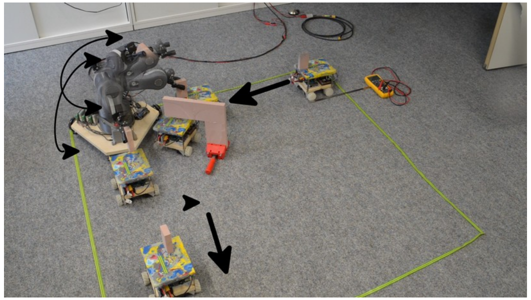
Fetch



Handover

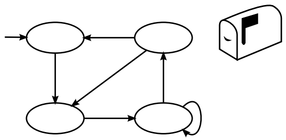


Twist



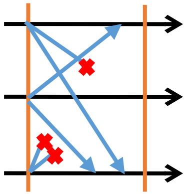
Underpass

Language to Help Decomposition



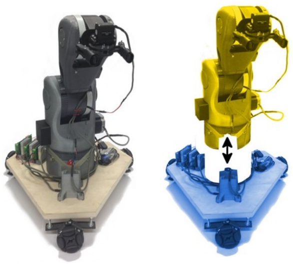
P

Isolates Synchronization from Data



PSync

Isolates Steps in a Distributed Algorithm



PGCD

Isolates Discrete and Continuous, Components