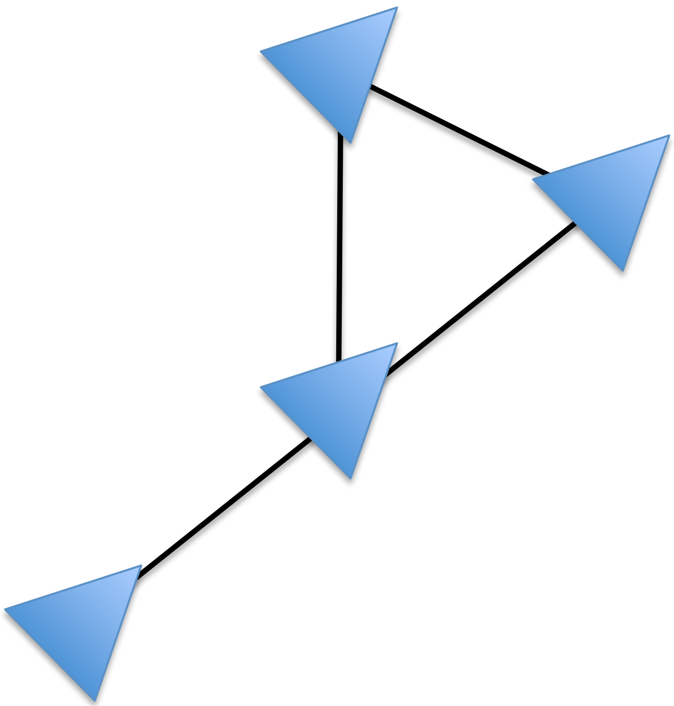
Specification and Programming of Networked Embedded Systems

Nils Napp
Fayette Shaw
Eric Klavins

University of Washington
Seattle, WA

Multi-Vehicle Control Systems



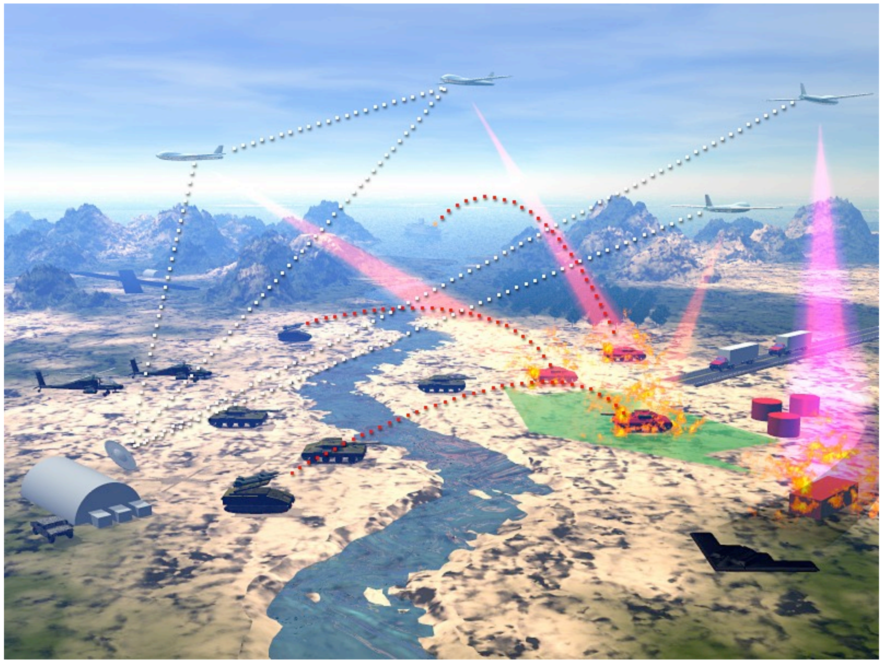
Example

$$\dot{x}_i = \sum_{j \in N(i)} x_j - x_i$$

Questions:

- How is x_i communicated?
- Are agents synchronized?
- What about switching to other tasks?
- Stochasticity?
- Faults?

Networked Embedded Systems



Asynchronous

Not necessarily “fair”

Complex software / mode switching / state

Control is a small part of the over engineered system

Guarded Command Languages

1975

Guarded Commands, Nondeterminacy and Formal Derivation of Programs

Edsger W. Dijkstra
Burroughs Corporation

So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be be shown.

Key Words and Phrases: programming languages, sequencing primitives, program semantics, programming language semantics, nondeterminacy, case-construction, repetition, termination, correctness proof, derivation of programs, programming methodology
CR Categories: 4.20, 4.22

```
q1, q2, q3, q4 := Q1, Q2, Q3, Q4;
do q1 > q2 → q1, q2 := q2, q1
▯  q2 > q3 → q2, q3 := q3, q2
▯  q3 > q4 → q3, q4 := q4, q3
od.
```

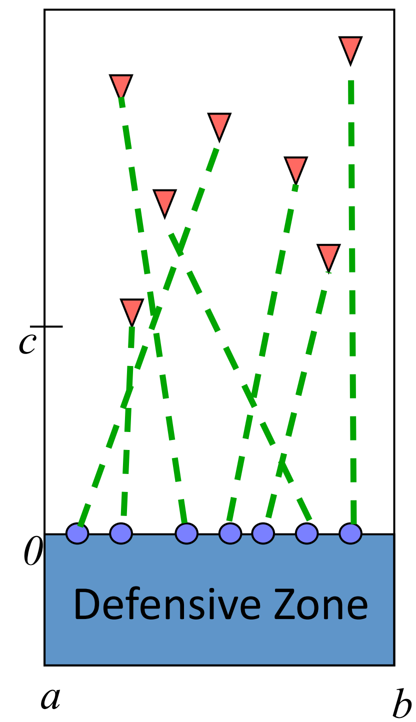
- Commands executed in any order.
- Proofs must show that all (fair) interleavings lead to the correct result
- Which is the same as showing global stability.
- (see also "self-stabilizing algorithms")

CCL: The Computation and Control Language

An Abstract CCL Program

$$
\begin{array}{r|l}
Red(i) & \\
\hline
\text{Initial} & x_i \in [a,b] \wedge y_i > c \\
\text{Commands} & y_i > \delta \;:\; y_i' = y_i - \delta \\
 & y_i \leq \delta \;:\; x_i' \in [a,b] \wedge y_i > c \\
Blue(i) & \\
\hline
\text{Initial} & z_i \in [a,b] \wedge z_i < z_{i+1} \\
\text{Commands} & z_i < x_{\alpha(i)} \wedge z_i < z_{i+1} - \delta \;:\; z_i' = z_i + \delta \\
 & z_i > x_{\alpha(i)} \wedge z_i > z_{i-1} + \delta \;:\; z_i' = z_i - \delta
\end{array}
$$

$$P_{Blue}(n) = +_{i=1}^{n} Blue(i)$$



CCLi = CCL Interpreter

- Guarded commands
- Program composition
- Strong type checker
- Extensible

Klavins, 2003.

Klavins and Murray, 2004.

Outline

- Fay
  - CCLi (The CCL Interpreter)
  - Example: Consensus with message passing
- Nils
  - CCLi with the "Factory Floor Testbed"
  - Composition
  - Stochastic Schedule

Introduction to CCL Syntax and Semantics

Fayette Shaw

16 September 2009

Distributed Systems

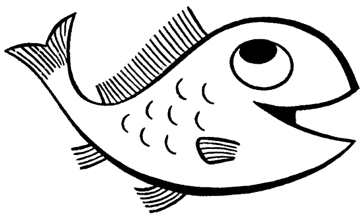



Outline

- Distributed Systems
- Overview of the language
  - Guarded commands
  - Extensibility and libraries
  - Programs and Composition
- Examples
  - Consensus

Guarded commands

- Local rules
- Guard, action

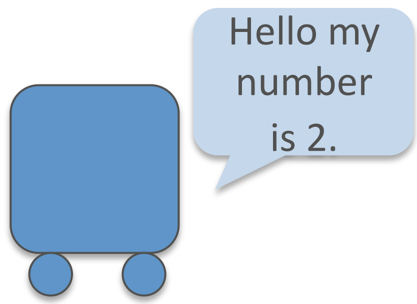


Global behavior

```
boolean expression :{
        command_1,
        command_2,
        …
        command_k
   };
```

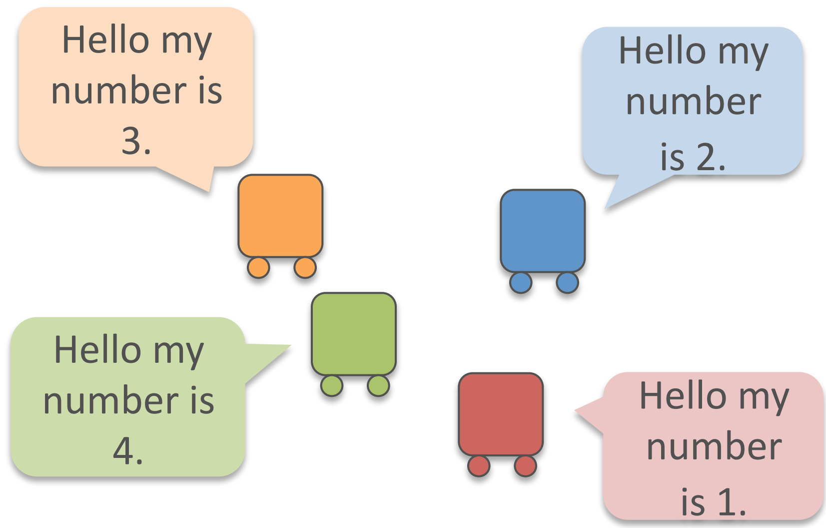
Guarded commands

- Local rules
- Guard, action




```
boolean expression :{
        command_1,
        command_2,
        …
        command_k
    };
```

Global behavior




```
true : {
   print ( "Hello my number
      is ", n ,"\n")
};
```

Programs

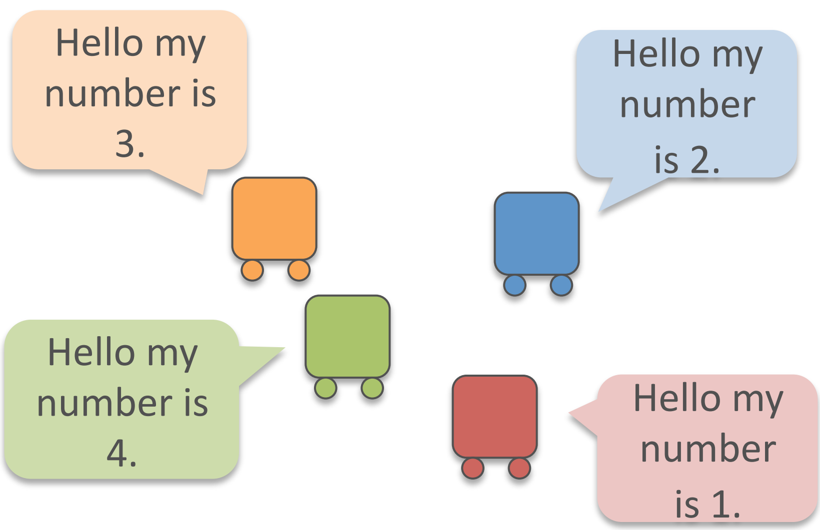
```
program p ( param_1, ..., param_n ) :={
    statement_1
    statement_2
    …
    statement_m
};
```

A statement is an initialization `x := 10` or a guarded command.

```
program agent ( n ) := {
  done := false;
  true : {
       print ( "Hello my number is ", n ,"\n"),
      done := true
  };
  done : { exit () };
};
```

Composition of programs

```
program agent ( n ) := {
    true : { print ( “Hello my number is ”, n ,”\n”) };
};

program main := agent (1) + agent (2) + agent (3) + agent (4);
```



in what order?

Execution order

```
program agent ( n ) := {
    true : { print ( "Hello my number is ", n ,"\n") };
};
program main := agent (1) + agent (2) + agent (3) + agent (4);
```

```
ccli system.ccl

Hello my number is 1
Hello my number is 2
Hello my number is 3
Hello my number is 4
Hello my number is 1
Hello my number is 2
Hello my number is 3
Hello my number is 4
Hello my number is 1
Hello my number is 2
Hello my number is 3
```

```
ccli system.ccl -r

Hello my number is 2
Hello my number is 3
Hello my number is 1
Hello my number is 4
Hello my number is 2
Hello my number is 4
Hello my number is 1
Hello my number is 3
Hello my number is 3
Hello my number is 2
Hello my number is 1
```

Types and Expressions

- Types
  - Boolean `true & false`
  - Integer/Real `( 1 + 1 ) ^ 5 + 7 % 3`
  - String `( "abc" <> "def” )`
  - List `1 @ { 2, 3 }`
  - Record `[to:= 1, from:= 7, msg:= “hi there!”]`
- Pre runtime type checker – no type errors!

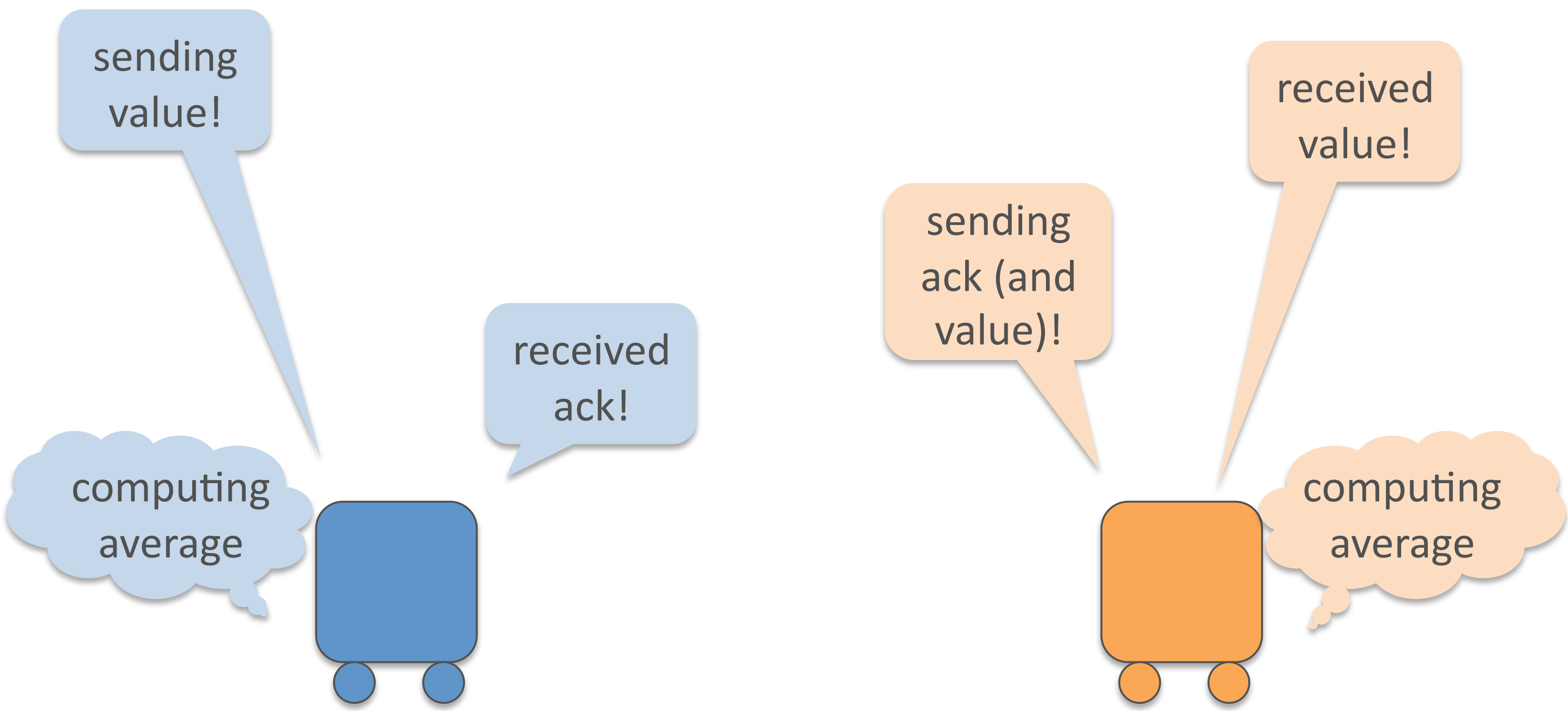
- Interesting Expressions
  - Conditional `if 1 > 0 then 1 else 0 end;`
  - Let `let x := 10, y := x/2 in x+y end;`
  - Lambda `f := lambda x . -x;`
  - Functions

```
fun fact n
  if n <= 0 then 1 else n* fact (n-1)
end;
```

Libraries

- Standard
- Math
- List
- Interprocess Communication
  - Mailboxes for easy concurrent programming
- UDP Datagrams
  - Multiple CCL programs talking over a network
- Graphics
  - Visualize dynamic data

Consensus


sending value!
received ack!
computing average
received value!
sending ack (and value)!
computing average

Inter-Process Communication

- iproc.ccl library

```
send([to:= 1, from:= 7])
```

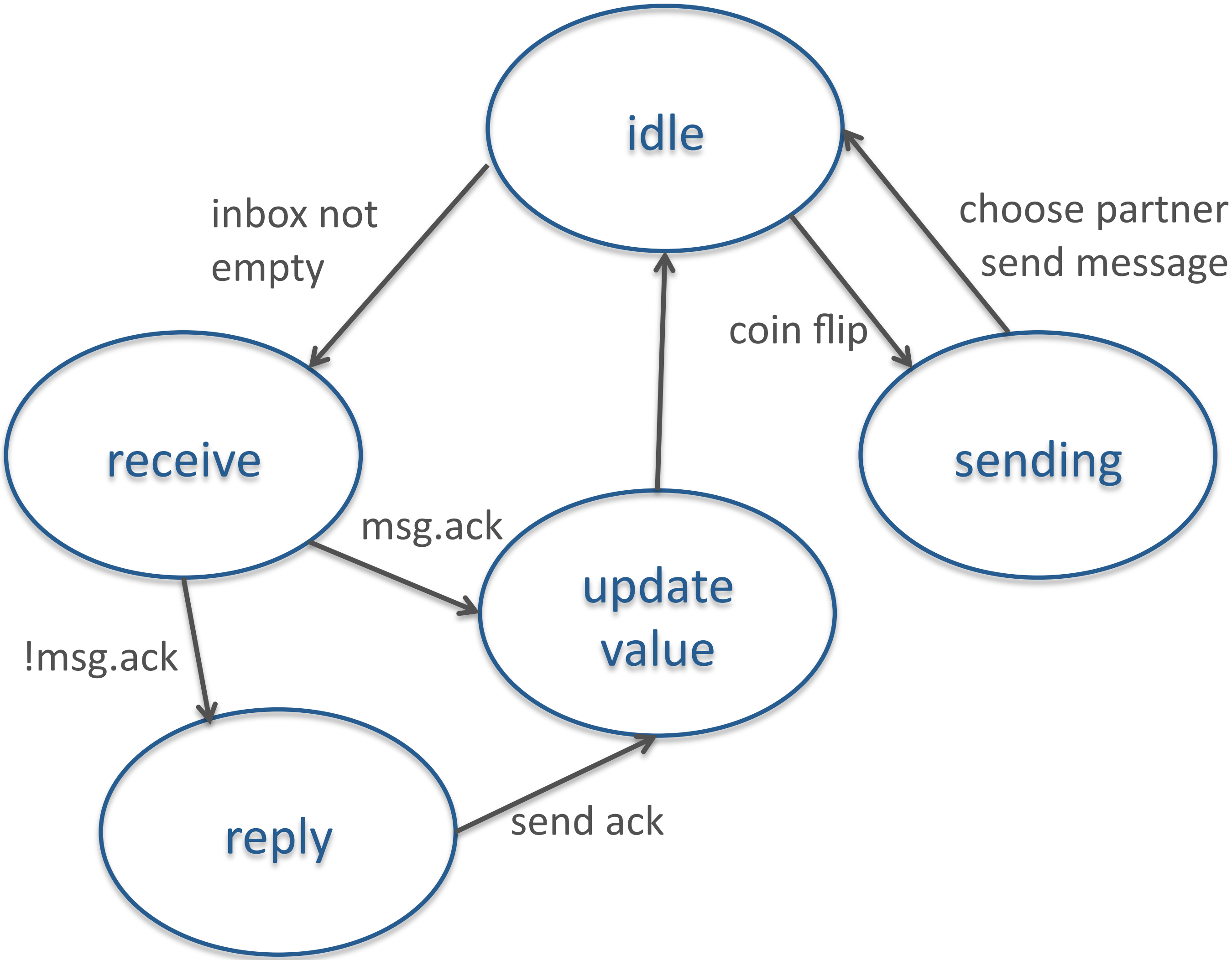
```
inbox(i)
```

returns boolean expression

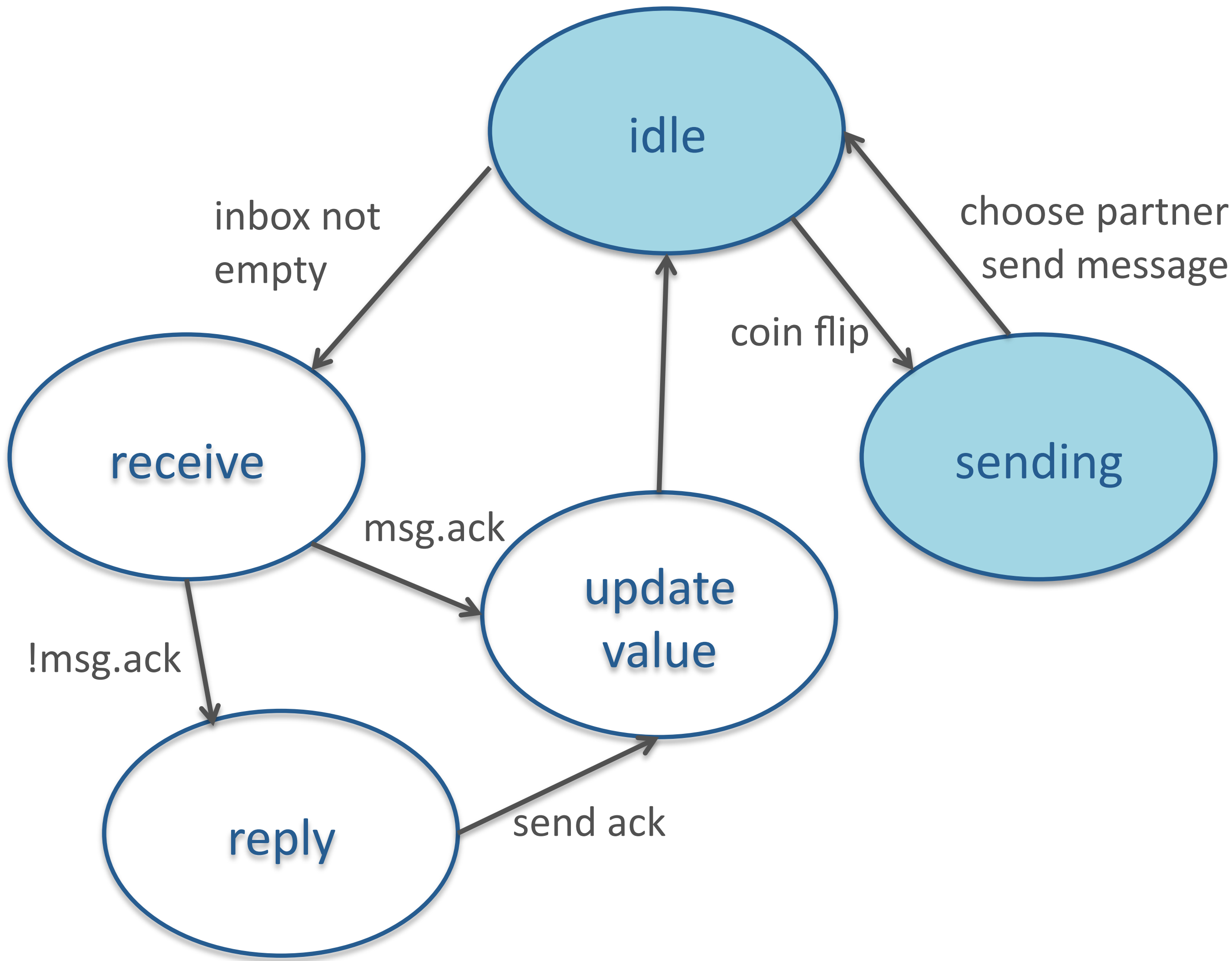
```
recv(i)
```

reads message

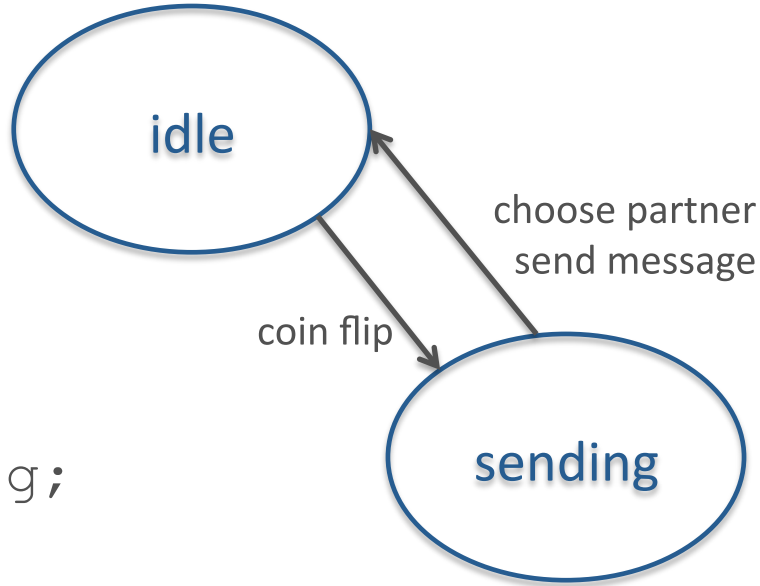
State Diagram



State Diagram

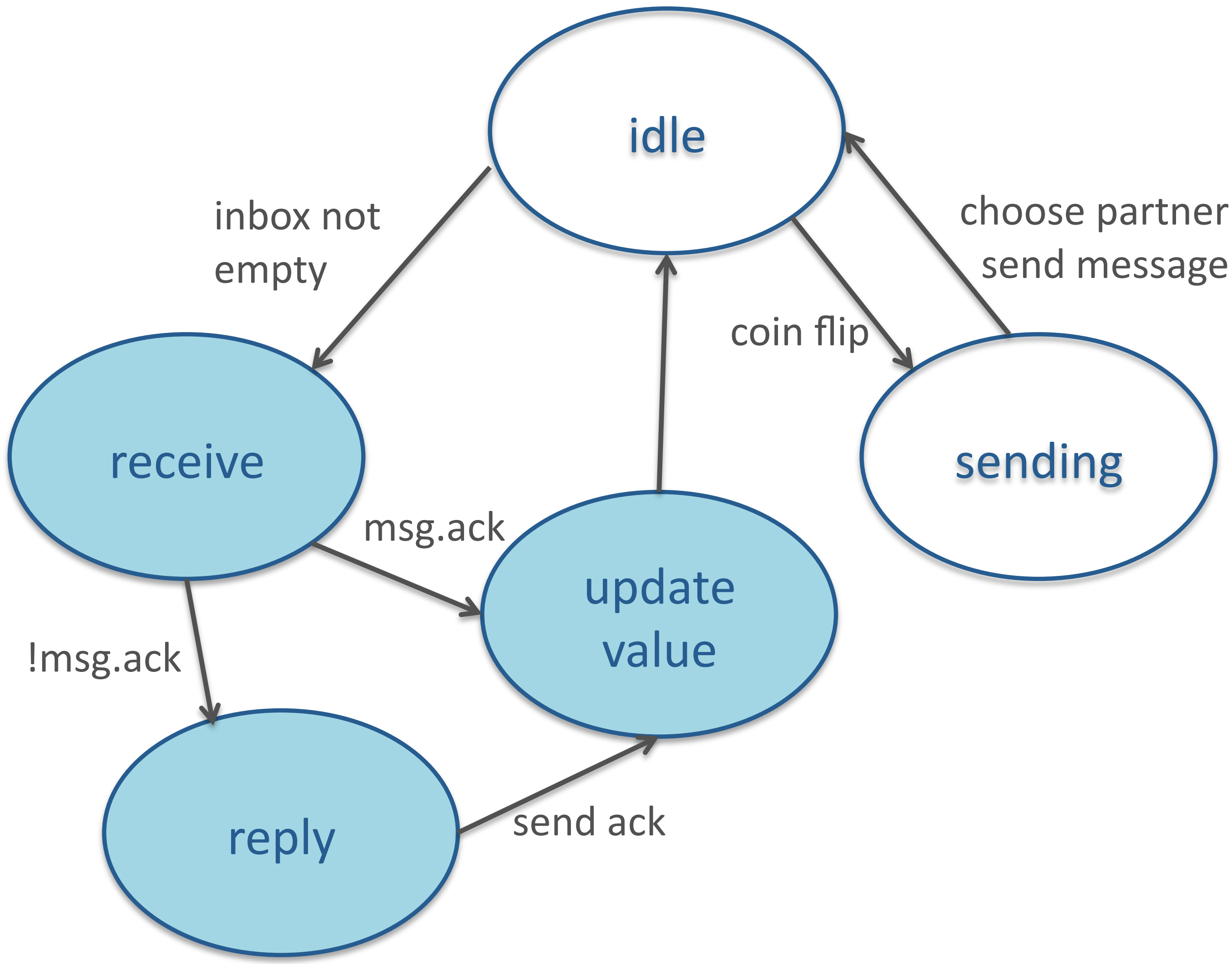


Send



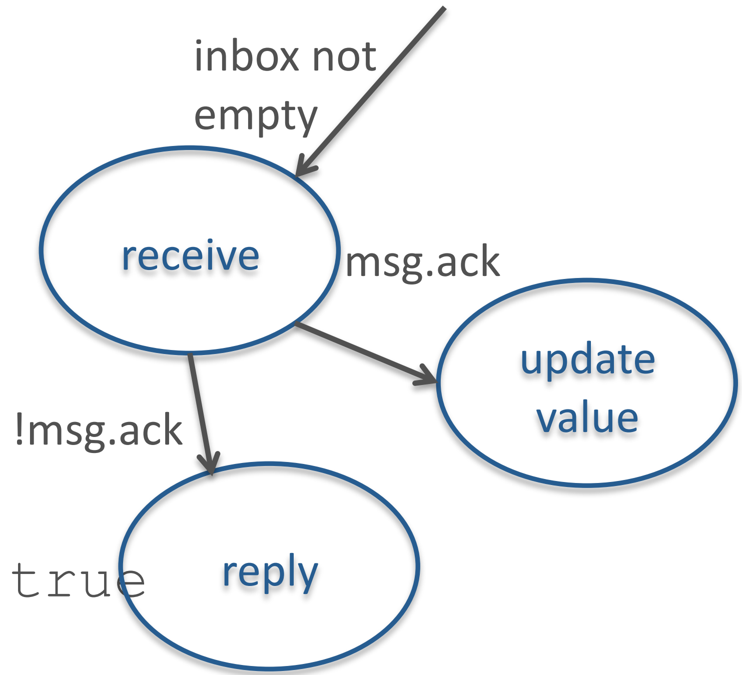

```
program send_msg (i) :={

  needs mode, x, toRobot, msg;

  (mode = "idle") & (rand(1000) = 1) :{
    toRobot := (i + rand( N - 1 ) + 1) % N,

    send([to := toRobot , from := i, x := x,
    ack := false]),

  };
};
```

State Diagram

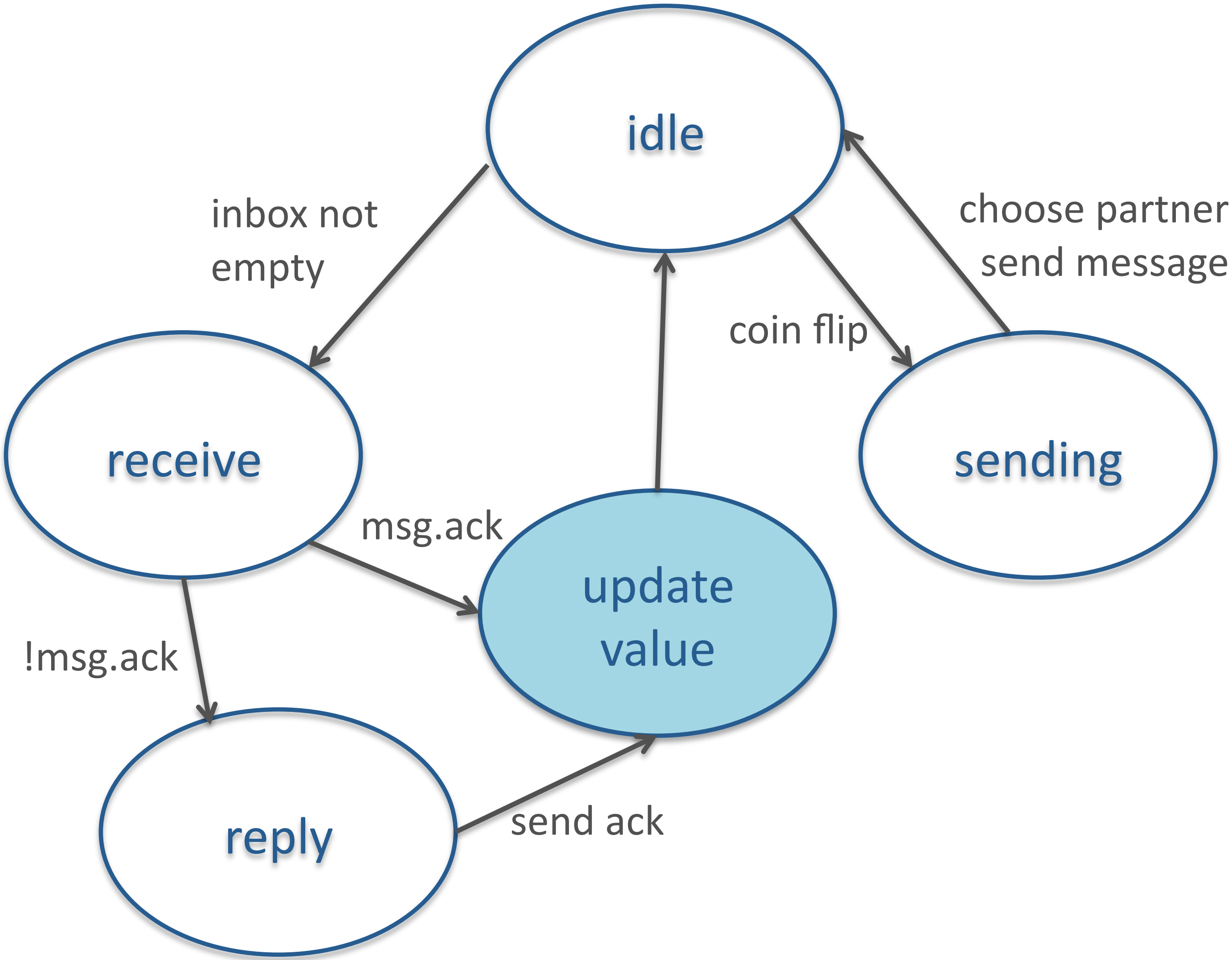


Receive

```
program recv_msg (i) :={
   needs x, mode, msg;
   inbox(i) :{
         msg := recv(i),

         mode := if msg.ack = true
             then "update"
             else "reply"
         end
    };
};
```



State Diagram



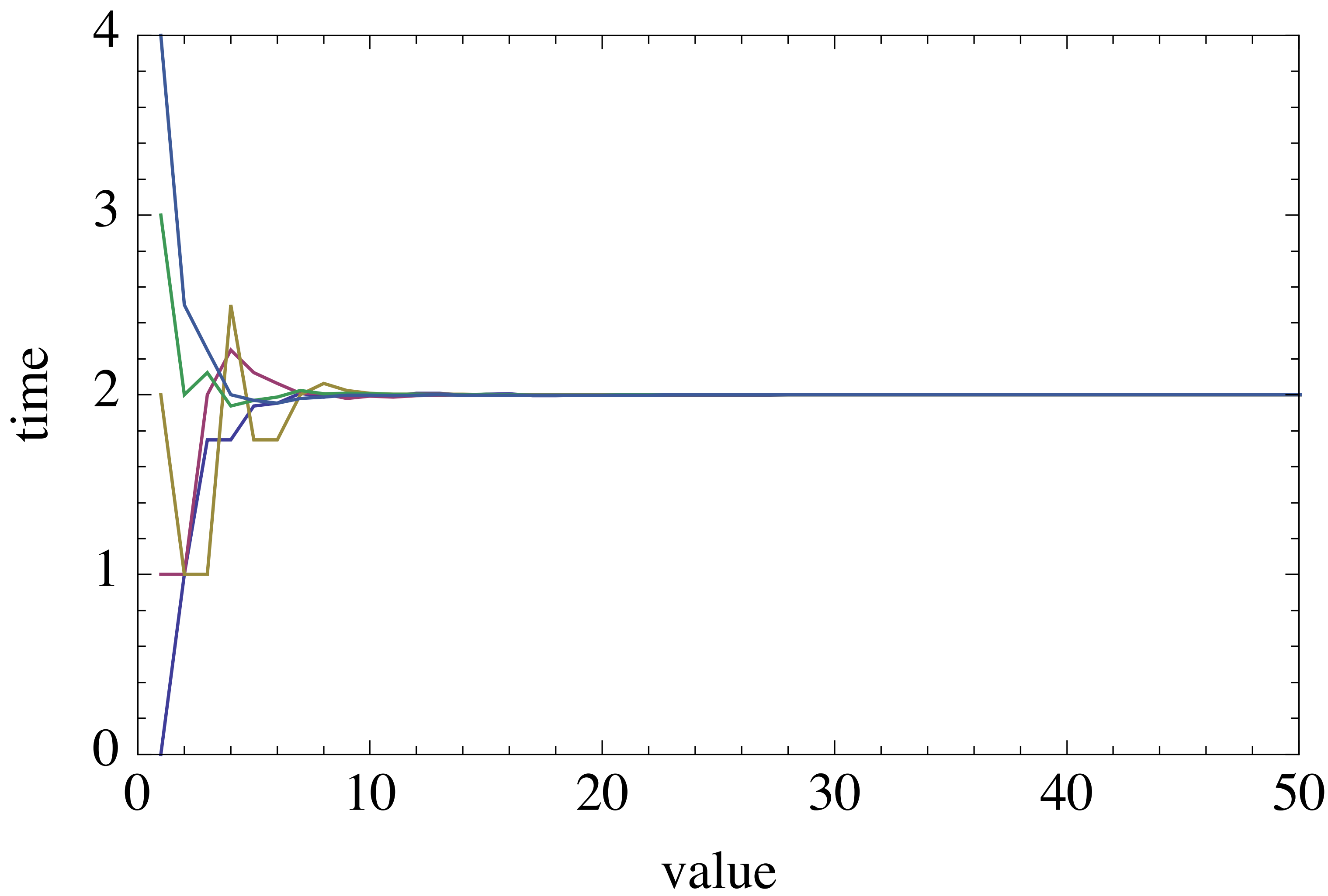
Process Message

```
program proc_msg (i) :={

   needs x, mode, msg;

   mode = "update" :{
        x:= 0.5*(msg.x+x),
        mode := "idle"
   };

   mode = "reply" :{
        send ([to:= msg.from, from:= i, x :=x,
   ack:=true]),
        mode := "update"
   };
};
```

Composition

```
program agent = send + receive
  sharing i x;

program main() := compose i in range
  n : agent ( i, n ) ;
```

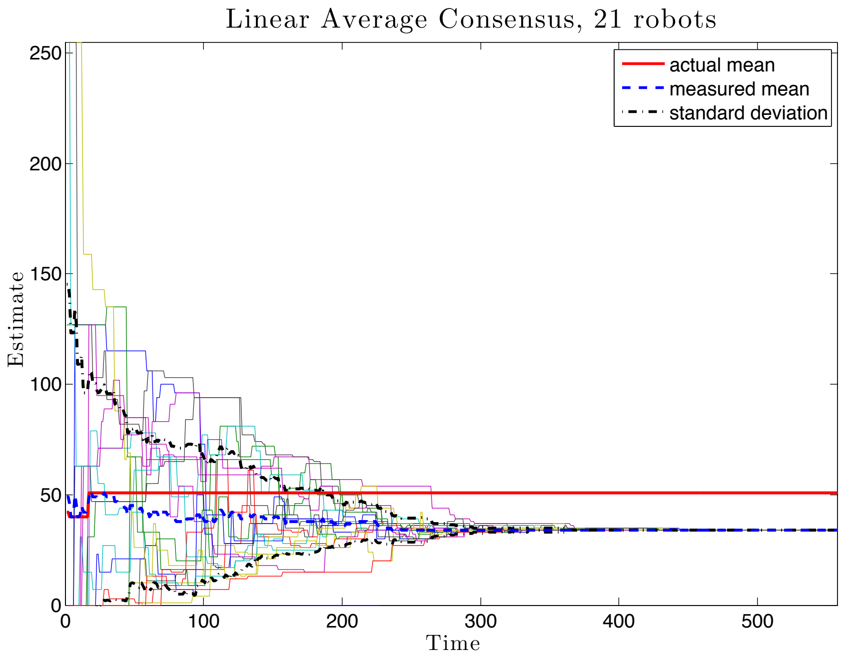
Plotted Data

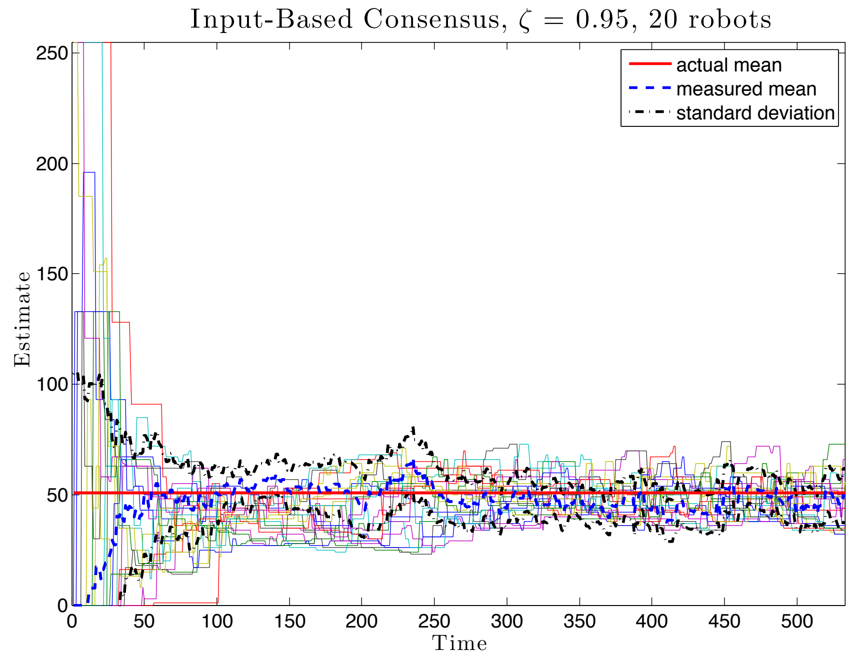


More Composition

```
program agent ( n ) = send + receive;

program interceptor = agent ( n ) +
  intercept;

program main() := compose i in range
  n : agent ( i ) + interceptor;
```

Dropped Messages





Extensibility

- `extern`
  - shared library

```
extern "C" Value * ccl_cos (list<Value *> * args ){
  return new Value ( cos ( ( *args -> begin())
    -> num_values() ) );
}
g++ -shared -o ccl_math.so ccl_math.cc -I
  \$(CCL_ROOT)/base
```

Software

- Available for
  - Mac
  - Linux

- source init_env
- make TARGET=MAC

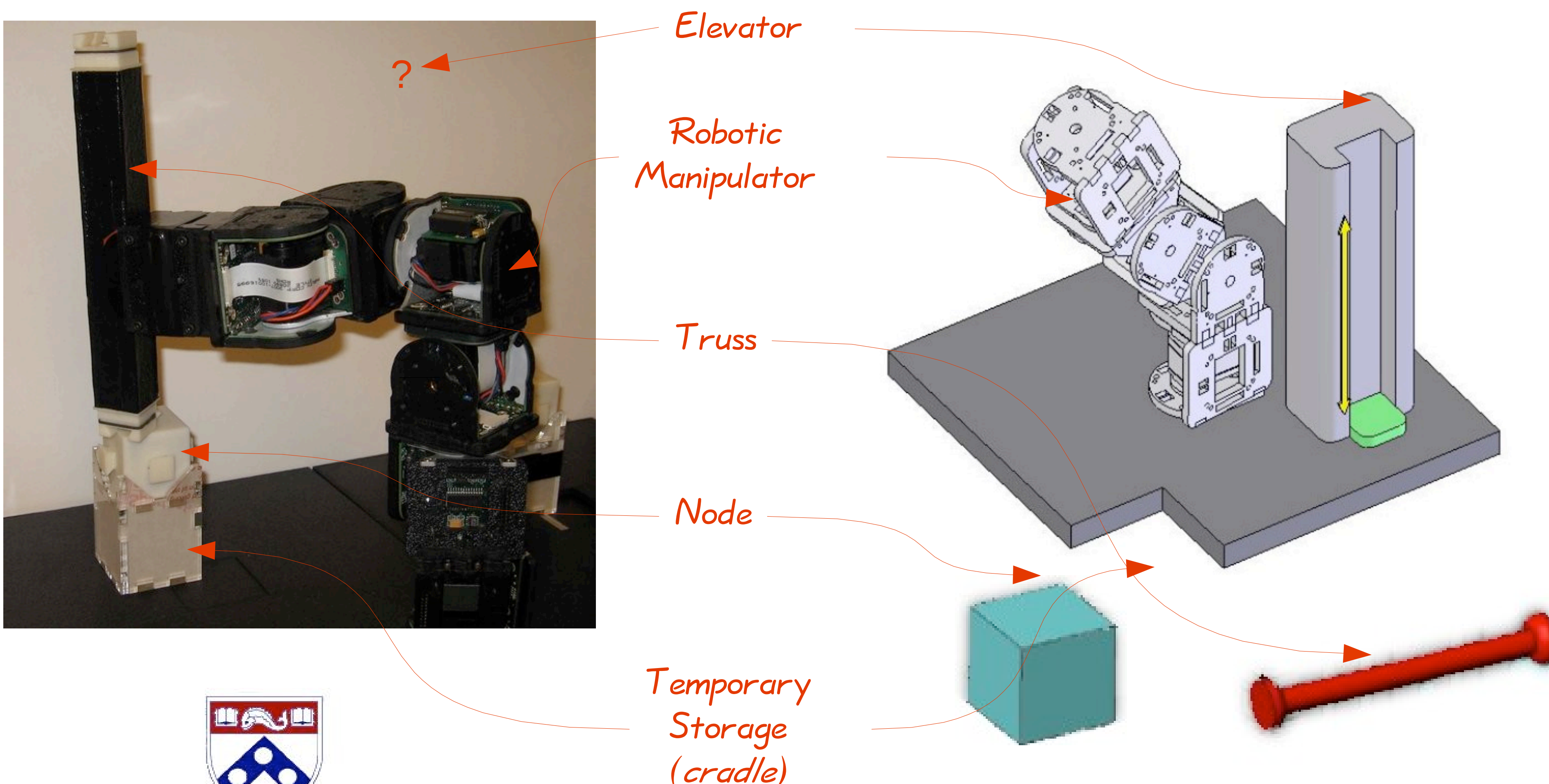
http://soslab.ee.washington.edu/mw/index.php/Code

The FF-Library for CCL

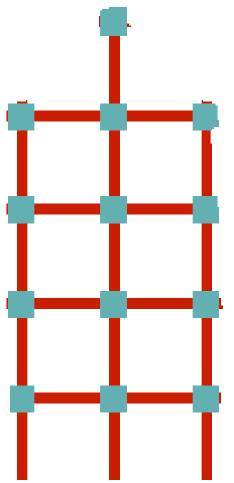
Nils Napp
Fayette Shaw
Eric Klavins

September 16th 2009
V&V MURI

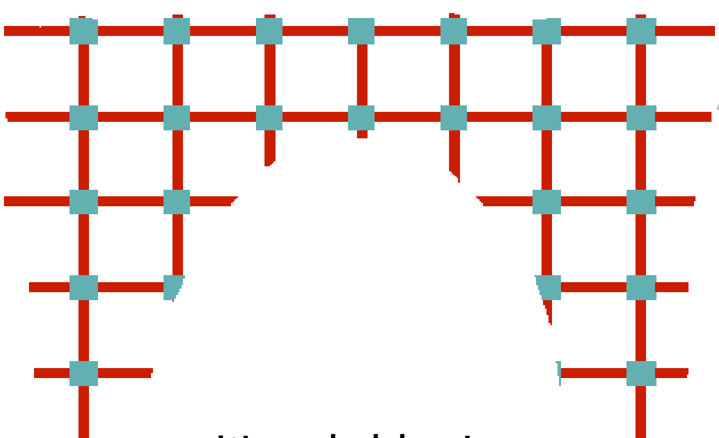
Stochastic Factory Floor (SFF)



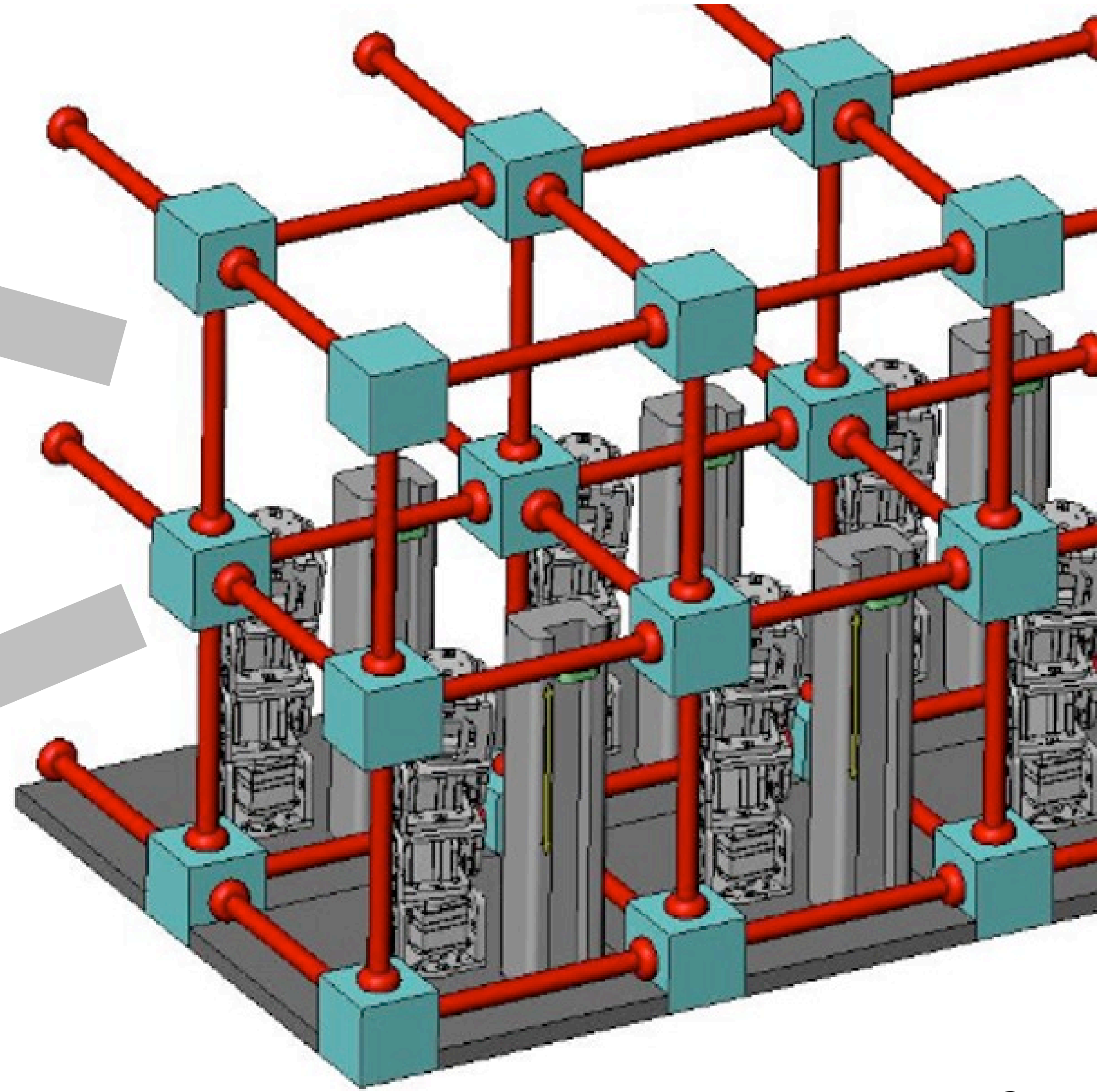
Stochastic Factory Floor (SFF)



It's a skyscraper!



It's a bridge!

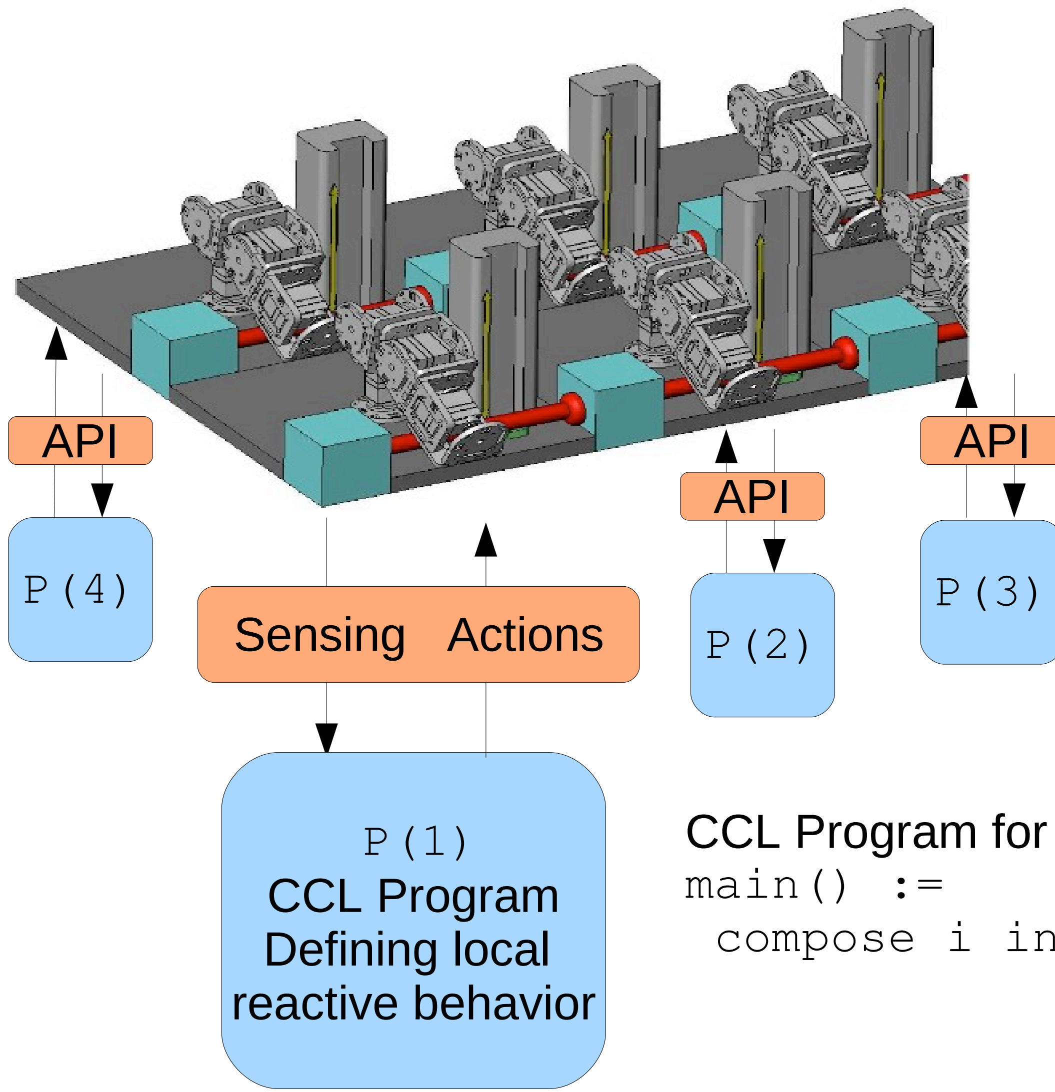


Reconfiguration Movie

Outline

- Motivation for using CCL
- API to Factory Floor Simulation
  - Localizing Functions
- Creating a Markov Process from a Program
  - Disassembly Program
- Routing programs
  - Fast vs. Robust Programs
  - Program "Robustification"
- Getting libff

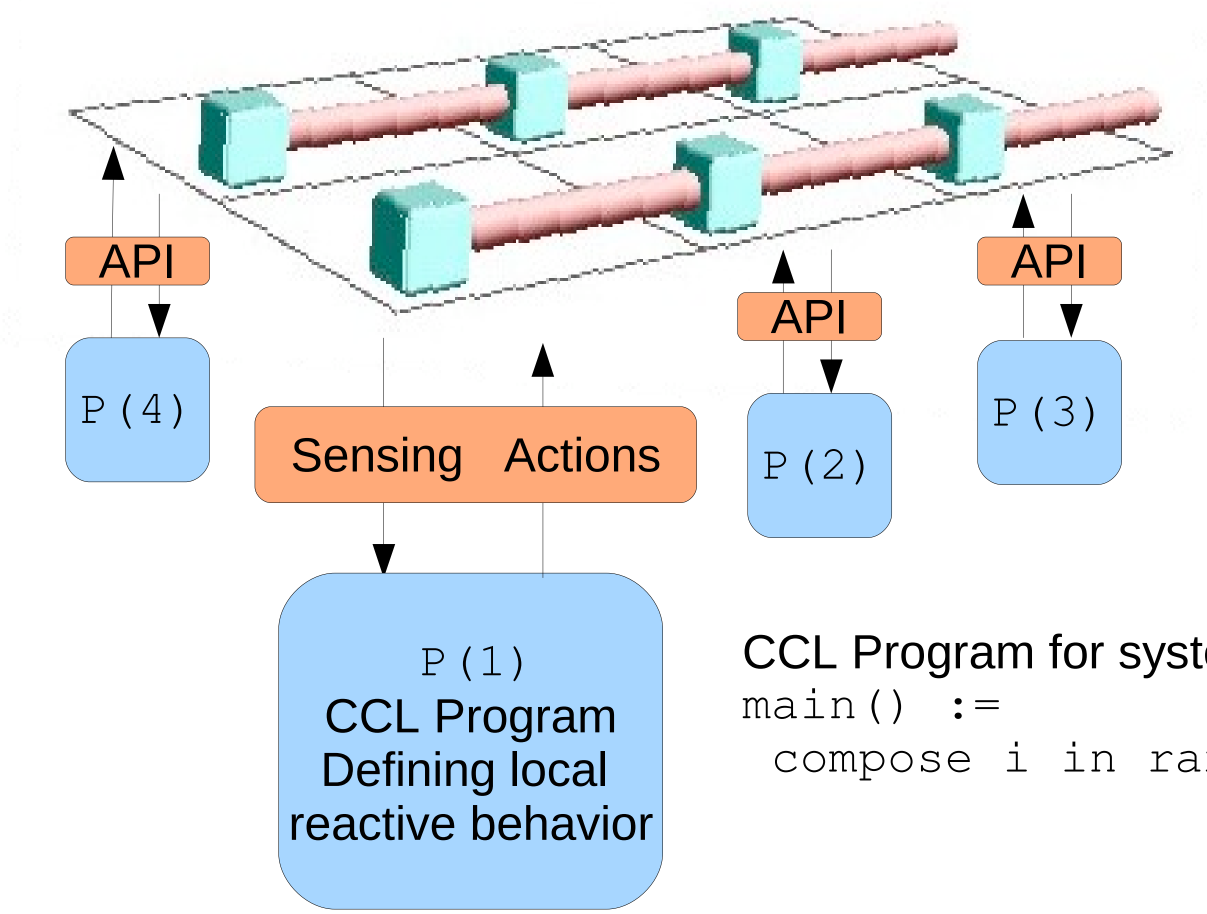
Programming the FF Testbed



CCL Program for system:

```
main() :=
 compose i in range 1 4: P (i);
```

Programming the FF Testbed



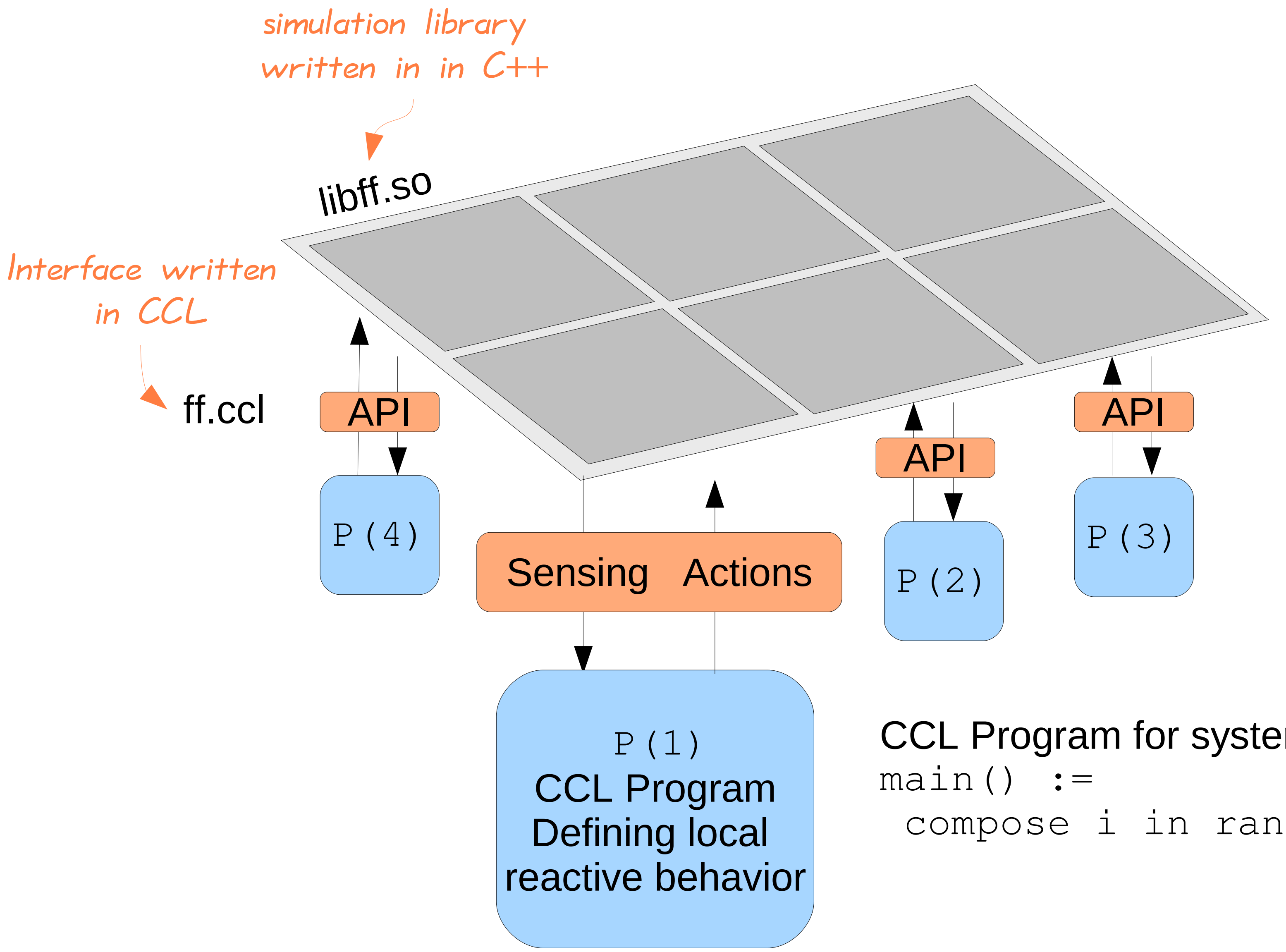
CCL Program for system:

```
main() :=
 compose i in range 1 4: P (i);
```

Outline

- Motivation for using CCL
- API to Factory Floor Simulation
  - Localizing Functions
- Creating a Markov Process from a Program
  - Disassembly Program
- Routing programs
  - Fast vs. Robust Programs
  - Program "Robustification"
- Getting libff

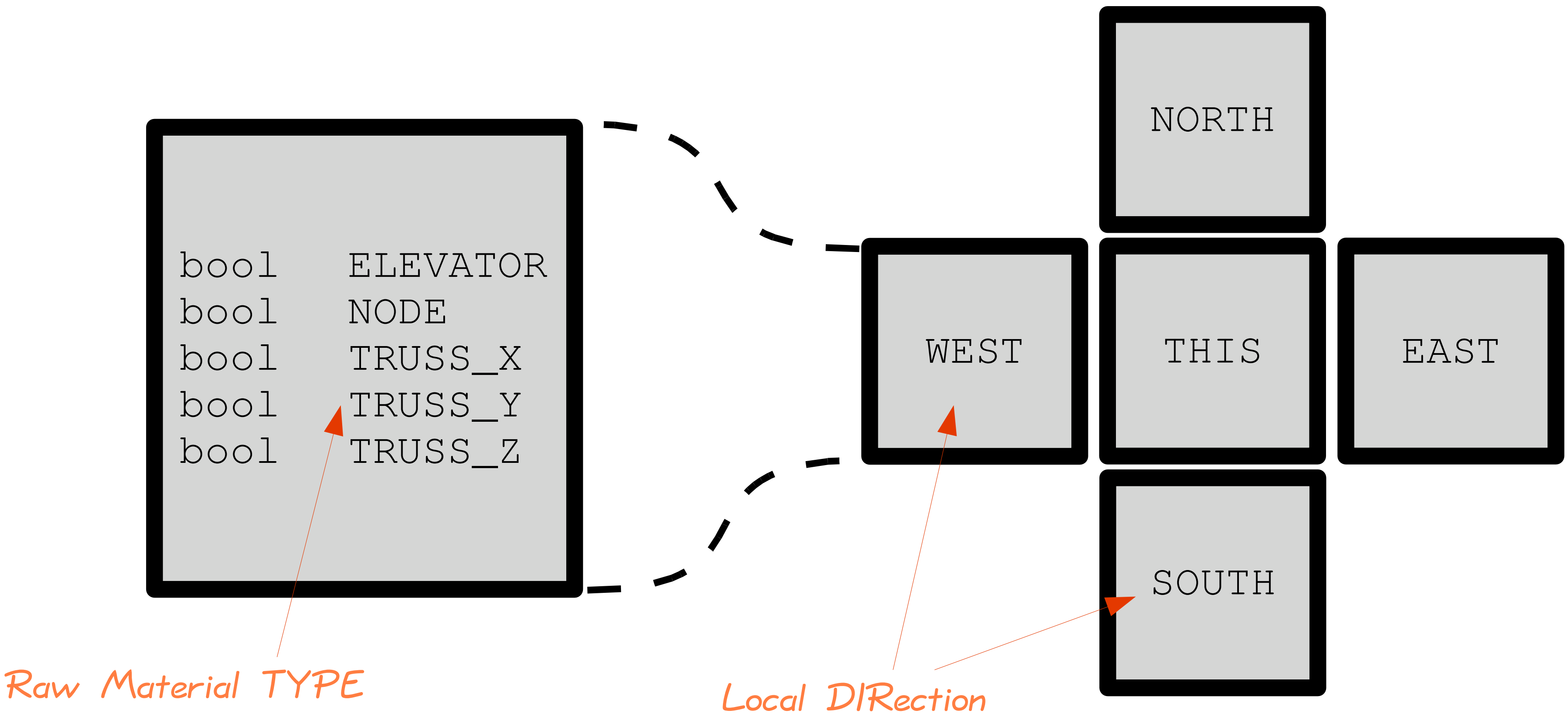
Programming the FF Testbed



CCL Program for system:

```
main() :=
 compose i in range 1 4: P (i);
```

Model of Factory Floor Modules



States of Factory Floor Module

Local Environment of Module

API to Factory Floor Modules

Directions (`DIR`) : `THIS, NORTH, EAST, SOUTH, WEST`
Type of Raw material (`TYPE`): `NODE, TRUSS_X, TRUSS_Y, TRUSS_Z`

<u>Checking status:</u>

```
checkFilled  DIR TYPE
checkEmpty   DIR TYPE
checkElevatorUp DIR
```

<u>Changing the state:</u>

```
moveNode  DIR
moveTruss TYPE DIR TYPE
insert TYPE
remove TYPE
lift
lower
```

API to Factory Floor Modules

From **ff.ccl**

```
…

/* for initializing simulation */
external unit initff() "libff.so" "ccl_FFinit";
external unit readStructure()  "libff.so" "ccl_FFread";

/* get sensor data */
external bool checkFilledXY(int, int, int, int) "libff.so"
"ccl_checkFilled";
external bool checkEmptyXY(int, int, int, int) "libff.so"
"ccl_checkEmpty";

/* act on on state */
external unit insertXY(int, int, int) "libff.so" "ccl_insert";
external unit removeXY(int, int, int) "libff.so" "ccl_remove";
external unit liftXY(int, int)  "libff.so" "ccl_lift";
external unit lowerXY(int, int) "libff.so" "ccl_lower";
external unit moveNodeXY(int,int, int) "libff.so" "ccl_moveNode";
external unit moveTrussXY(int, int, int, int ,int) "libff.so"
"ccl_moveTruss";
...
```

Outline

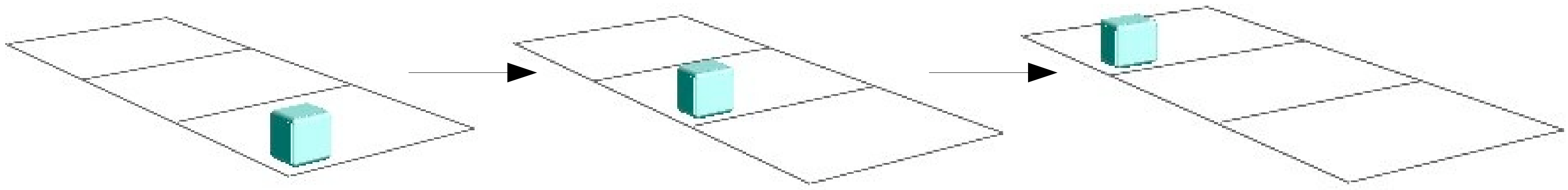
- Motivation for using CCL
- API to Factory Floor Simulation
  - Localizing Functions
- Creating a Markov Process from a Program
  - Disassembly Program
- Routing programs
  - Fast vs. Robust Programs
  - Program “Robustification”
- Getting libff

Example: `moveNode`

```
...
external bool checkFilledXY(int, int, int, int) "libff.so" "ccl_checkFilled";
external bool checkEmptyXY(int, int, int, int) "libff.so" "ccl_checkEmpty";
...
external unit moveNodeXY(int,int, int) "libff.so" "ccl_moveNode";
...
```

```
include ff.ccl

program north(x,y):= {
    checkFilled(x,y,THIS,NORTH) & checkEmpty(x,y, NORTH, NODE):{
        moveNode(x,y,NORTH);
    };
};

program main():=  north(0,1) + north(0,1) + north(0,2);
```

Don't want global coordinate



Localize API

From **ffFun.ccl**

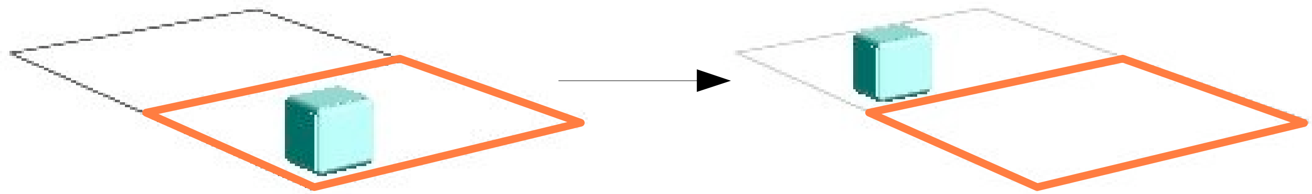
```
…

checkFilled :=
   (lambda dir. ( lambda type. checkFilledXY(x,y,dir,type)));

checkEmpty :=
   (lambda dir. ( lambda type. checkEmptyXY(x,y,dir,type)));

moveNode := ( lambda dir. moveNodeXY(x,y,dir));

...
```

```
include ff.ccl

program north(x,y):= {
    include ffFun.ccl

    (checkFilled THIS NORTH) & (checkEmpty NORTH NODE):{
        MoveNode NORTH;
    };
};
```

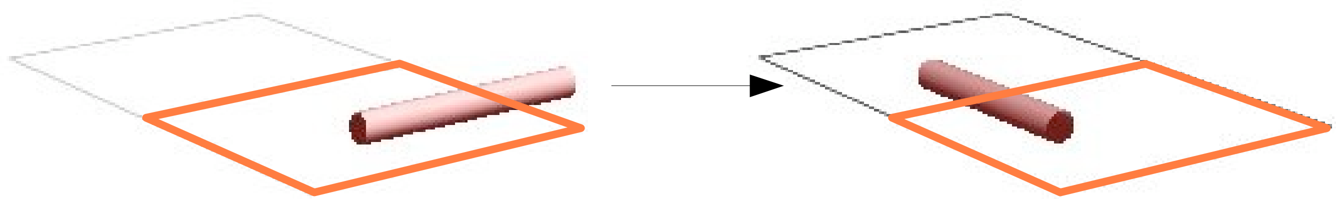
This is purely local

Example: Moving Materials

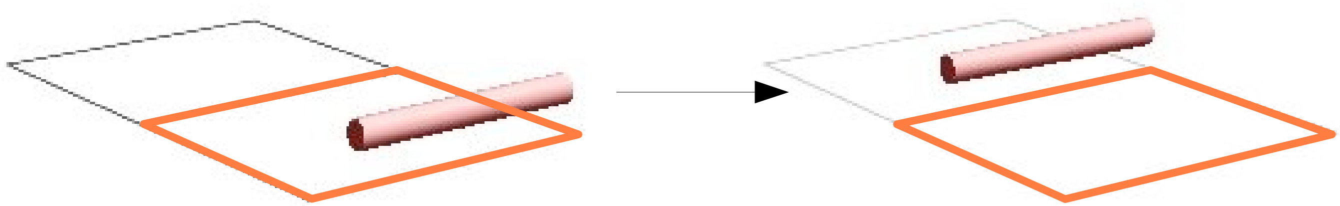
```
(checkFilled THIS NODE ) & (checkEmpty NORTH NODE) : {
   moveNode NORTH
};
```



```
(checkFilled THIS TRUSS_X ) & (checkEmpty THIS TRUSS_Y) : {
   moveTruss TRUSS_X THIS  TRUSS_Y
};
```



```
(checkFilled THIS TRUSS_X ) & (checkEmpty NORTH TRUSS_X) : {
   moveTruss TRUSS_X NORTH TRUSS_X
};
```



Outline

- Motivation for using CCL
- API to Factory Floor Simulation
  - Localizing Functions
- Creating a Markov Process from a Program
  - Disassembly Program
- Routing programs
  - Fast vs. Robust Programs
  - Program “Robustification”
- Getting libff

Turing a Programs into Markov Processes

Why?

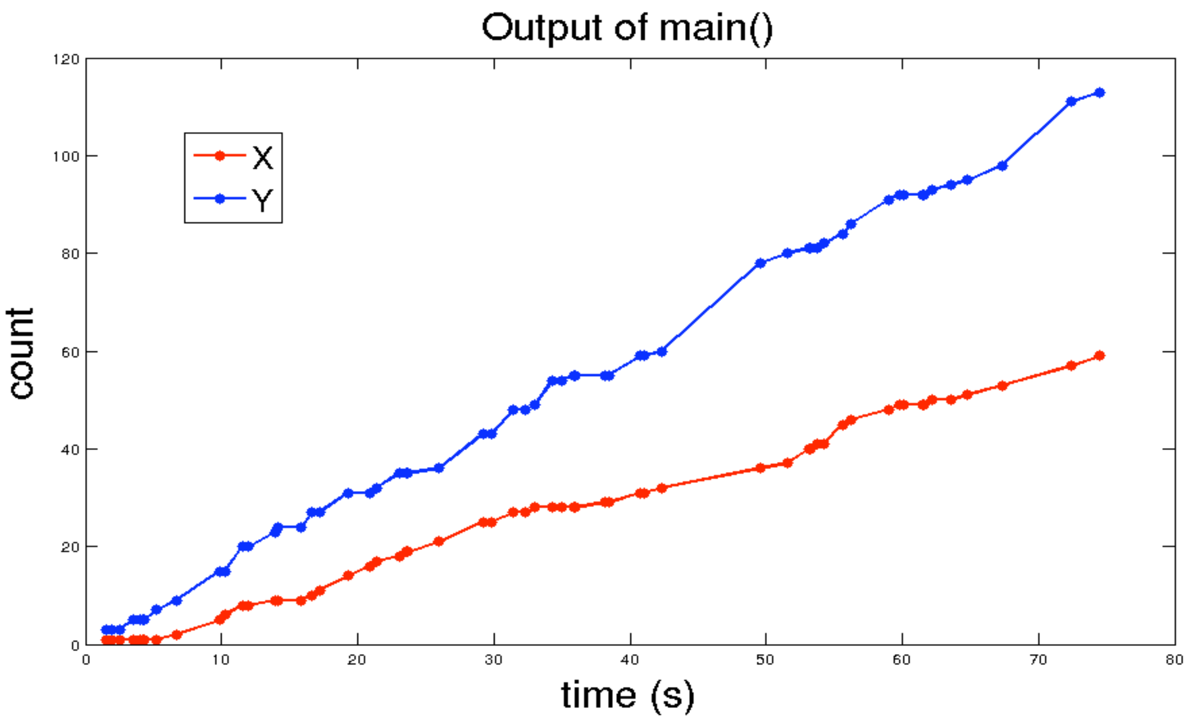
- Can analyze probabilistic failures
- Can use powerful tools from Markov Processe
- Allows to be robust to certain kinds of failues.

How?

- Add a *rate* to each guarded command
- Result is a Markov process on the state space of the program

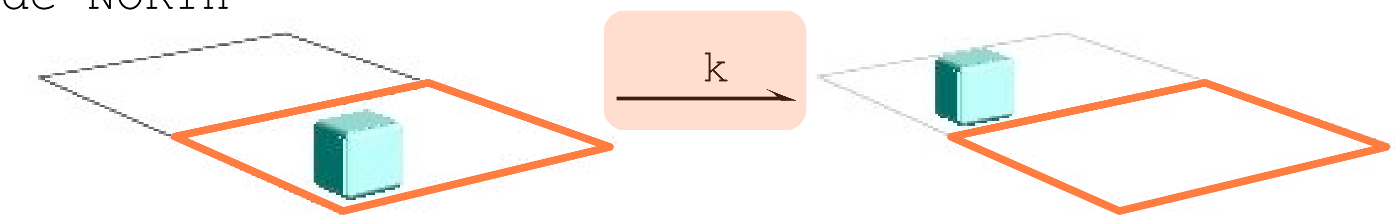
The `rate` guard

```
program drunkard( k ) := {
     X:=0;
     (rate k):{ X++ };
  };

program main() := updateDT() + drunkard( 1.0 ) + drunkard( 2.0 );

main() = drunkard(1.0) + drunkard(2.0) + updateDT();
```

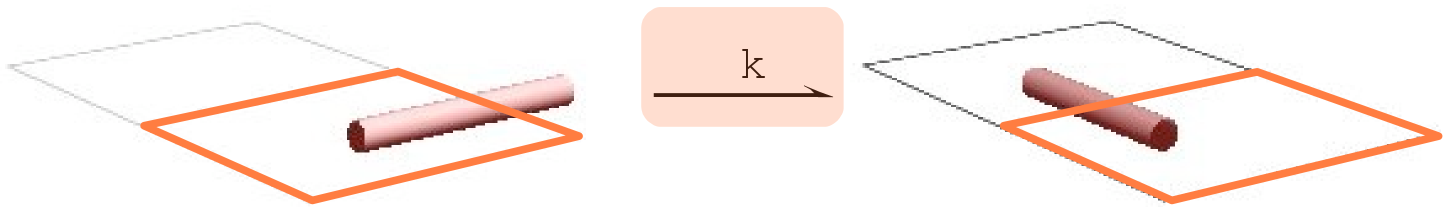


Example: Moving Materials

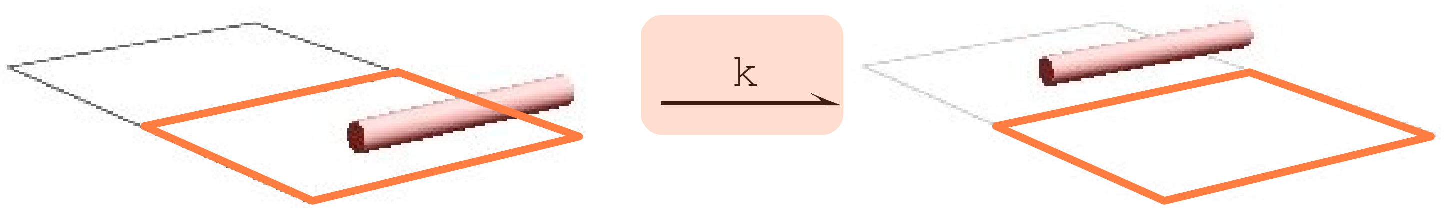
```
(rate k) & (checkFilled THIS NODE) & (checkEmpty NORTH NODE):{
   moveNode NORTH
};
```




```
(rate k) & (checkFilled THIS TRUSS_X) & (checkEmpty THIS TRUSS_Y):{
   moveTruss TRUSS_X THIS  TRUSS_Y
};
```




```
(rate k) & (checkFilled THIS TRUSS_X) & (checkEmpty NORTH TRUSS_X):{
   moveTruss TRUSS_X NORTH TRUSS_X
};
```



Outline

- Motivation for using CCL
- API to Factory Floor Simulation
  - Localizing Functions
- Creating a Markov Process from a Program
  - Disassembly Program
- Routing programs
  - Fast vs. Robust Programs
  - Program “Robustification”
- Getting libff

Example: Disassembling Structures

```
program disassembleXY(x,y):={

    include ffFun.ccl

    (rate (kglobal)) & (checkFilled THIS NODE) & (checkEmpty EAST NODE):{
        moveNode EAST
    };
    (rate (kglobal)) & (checkFilled THIS TRUSS_X) & (checkEmpty EAST TRUSS_X):{
        moveTruss TRUSS_X  EAST TRUSS_X
    };
    (rate (kglobal)) & (checkFilled THIS TRUSS_Y) & (checkEmpty EAST TRUSS_Y):{
        moveTruss TRUSS_Y EAST TRUSS_Y
    };

    (rate (kglobal)) & (checkFilled THIS TRUSS_Z) & (checkEmpty THIS TRUSS_Y)
            & (checkEmpty THIS TRUSS_X) & (checkEmpty THIS NODE)
       & !(checkFilled WEST TRUSS_Z) & !(checkFilled WEST TRUSS_Y):{
        moveTruss TRUSS_Z THIS TRUSS_Y,
        lift
    };
    (rate (kglobal)) & (checkEmpty THIS TRUSS_Y) & (checkElevator THIS): {
          lower
    };
};

program main() := disassembleXY(0,0) + disassembleXY(0,1) + disassembleXY(0,2)
                  disassembleXY(1,0) + disassembleXY(1,1) + disassembleXY(1,2);
```

Move stuff out of way

remove Z-truss & use elevator

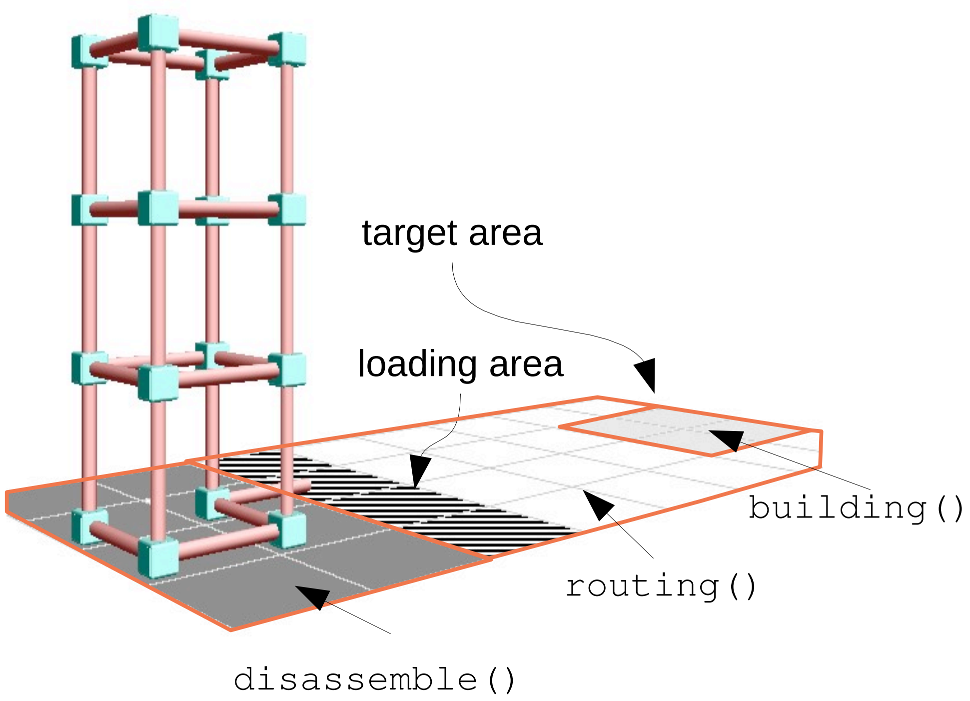
Example: Disassembling Structures

eat()

disassemble()

`main():= disassemble() + eat();`

Reconfiguration Program

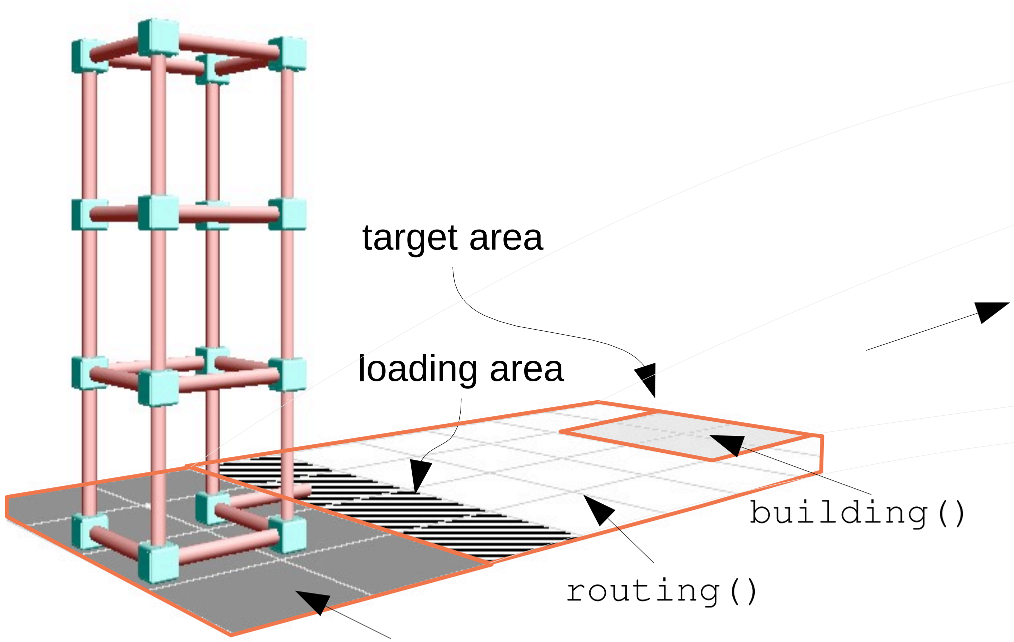


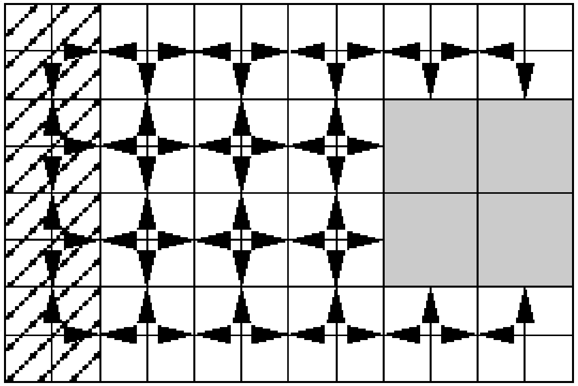
main():= disassemble() + routing() + building();

Outline

- Motivation for using CCL
- API to Factory Floor Simulation
  - Localizing Functions
- Creating a Markov Process from a Program
  - Disassembly Program
- Routing programs
  - Fast vs. Robust Programs
  - Program “Robustification”
- Getting libff

Routing Program



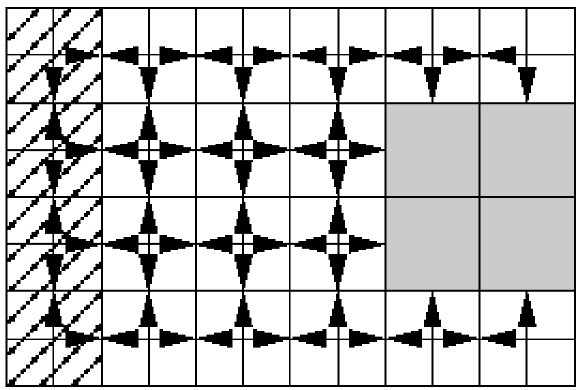


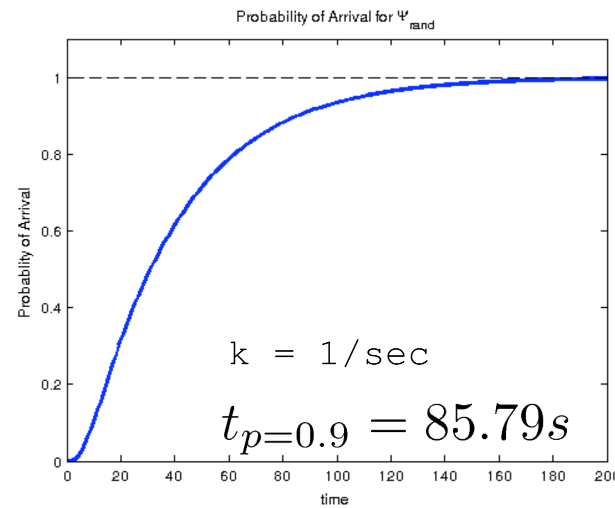
Quality of Routing Program:

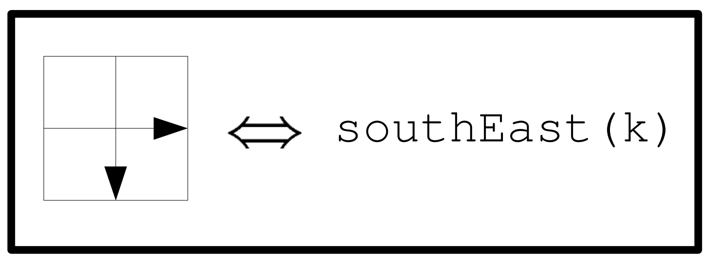
- Part appears in loading area at time 0
- Prob(target area at time t)

```
main():= disassemble() + routing() + building();
```

Random Walk: `random(k)`

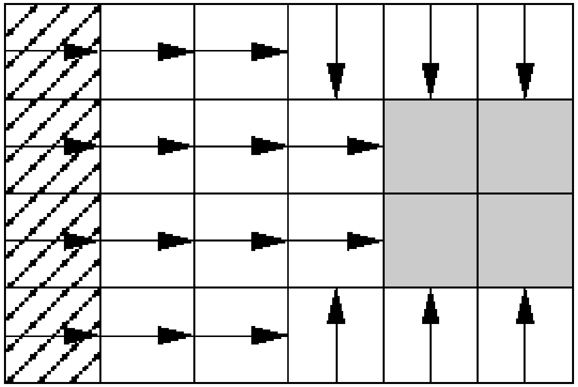


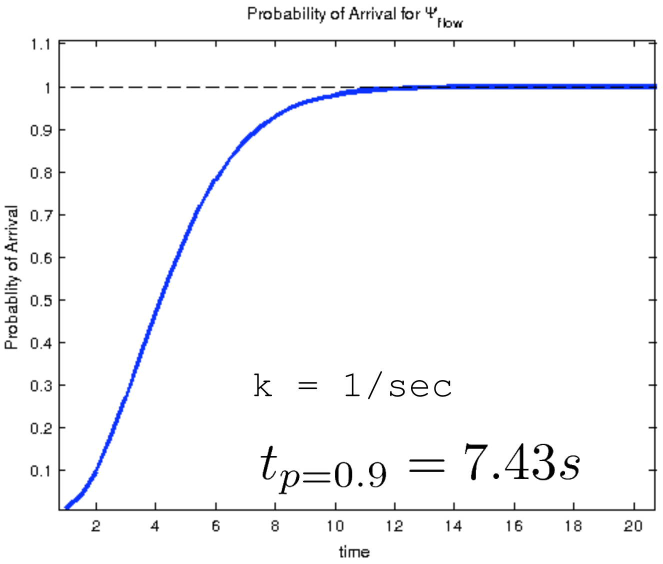


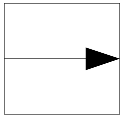



```
program southEast(k) := {
     (rate (k / 2)) & (checkFilled THIS NODE) & (checkEmpty EAST NODE):{
         moveNode EAST
     };
     (rate (k / 2)) & (checkFilled THIS NODE) & (checkEmpty SOUTH NODE):{
         moveNode SOUTH
     };
};
```

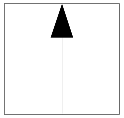
Flow Field: `flow(k)`



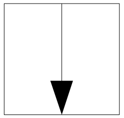




```
program east(k):= {
    (rate k ) & (checkFilled THIS NODE) & (checkEmpty EAST NODE):{
        moveNode EAST
    };};
```



```
program north(k):= {
    (rate k ) & (checkFilled THIS NODE) & (checkEmpty NORTH NODE):{
        moveNode NORTH
    };};
```

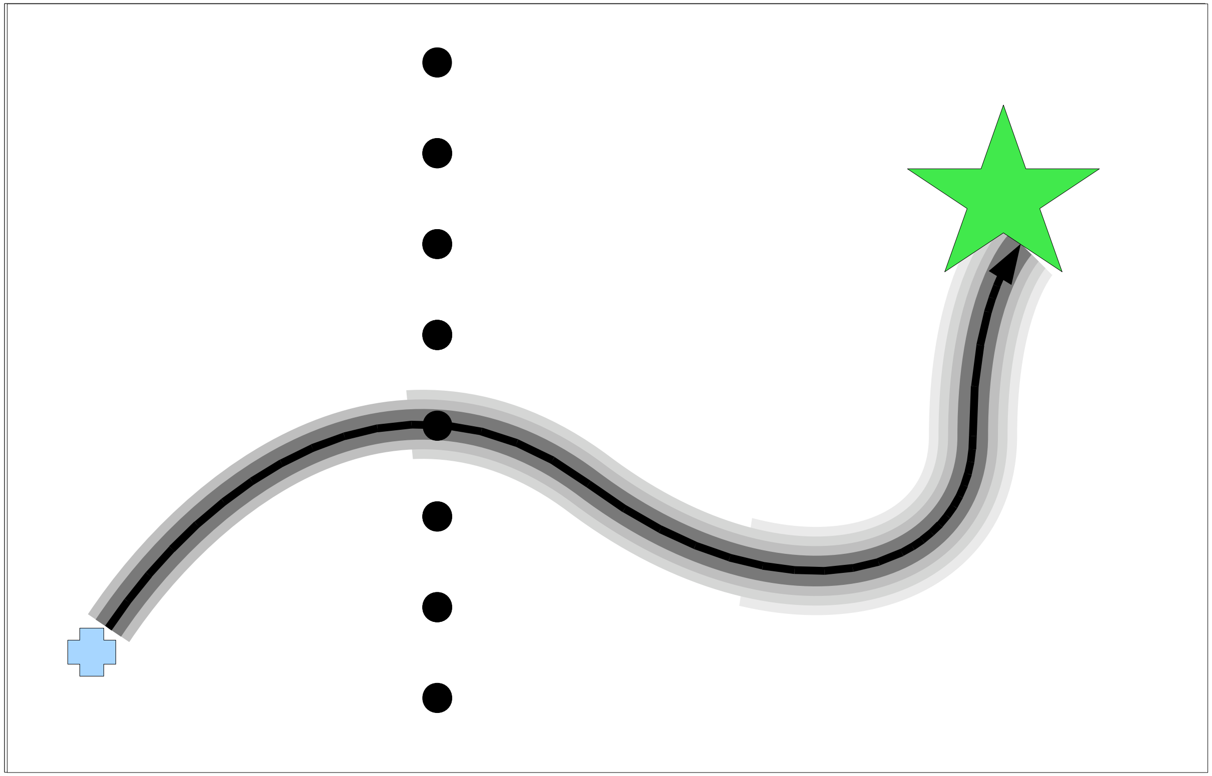


```
program south(k):= {
    (rate k ) & (checkFilled THIS NODE) & (checkEmpty SOUTH NODE):{
        moveNode SOUTH
    };};
```

Outline

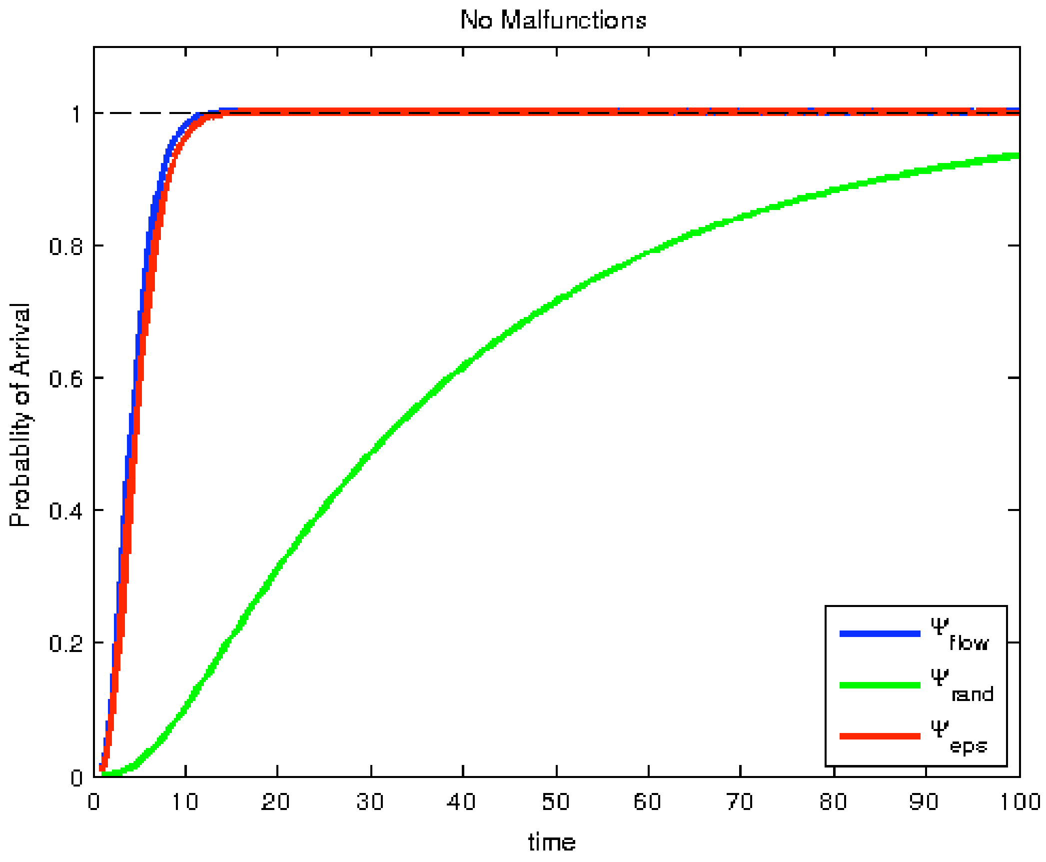
- Motivation for using CCL
- API to Factory Floor Simulation
  - Localizing Functions
- Creating a Markov Process from a Program
  - Disassembly Program
- Routing programs
  - Fast vs. Robust Programs
  - Program “Robustification”
- Getting libff

Path to Success



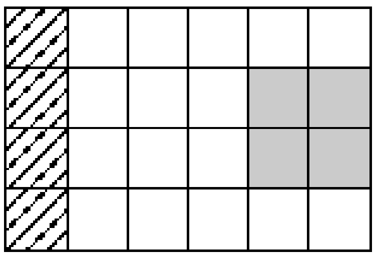
Robustifying Programs

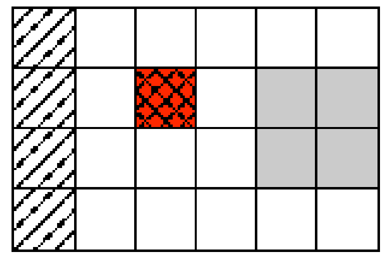
```
robust (k) := flow( 0.9*k )  + random (0.1 * k);
```

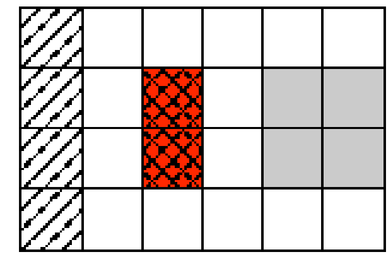


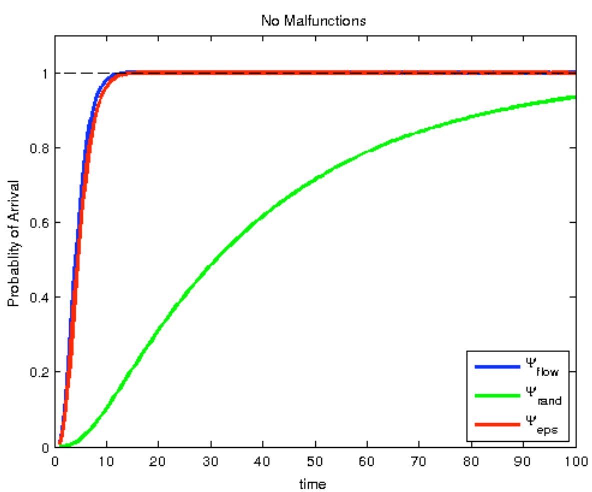
flow(1)   random(1)   robust(1)

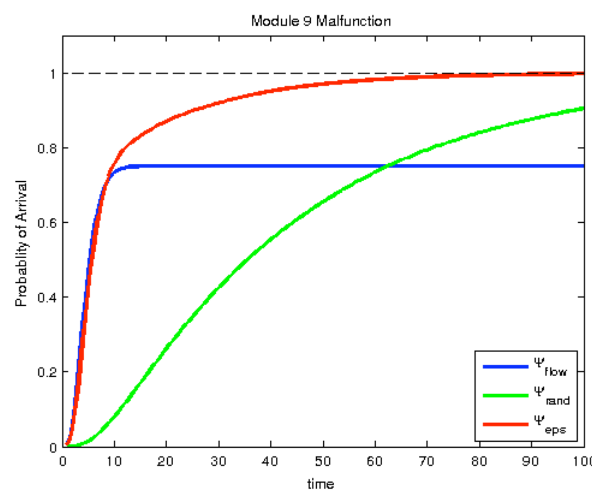
Robustness vs. Performance

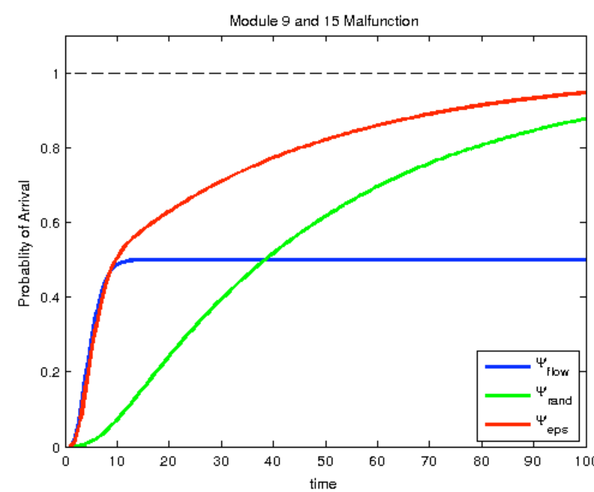








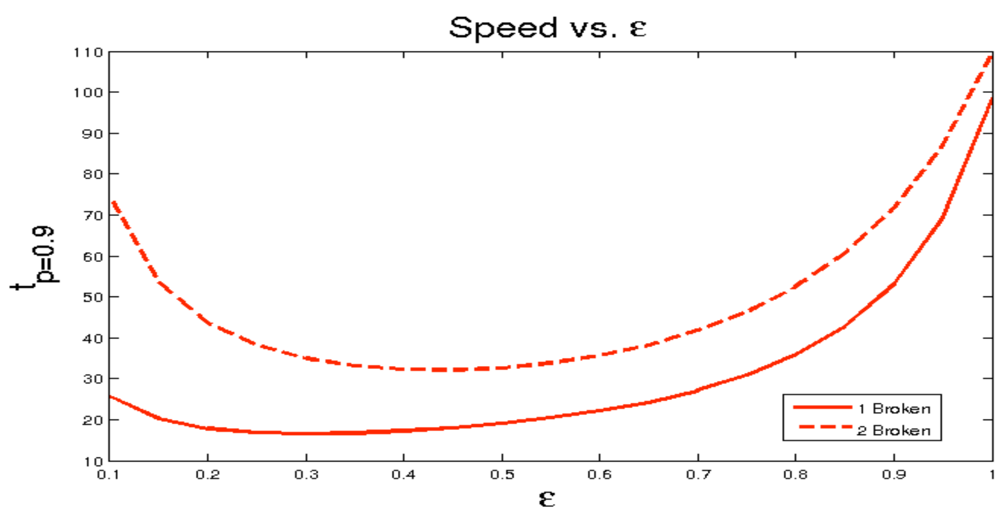




flow(1) random(1) robust(1)

Analyzing the Markov Process

		Ψ_{Flow}	Ψ_{Random}	$\Psi_{\epsilon}\ \epsilon = 0.1$
$t_{P=0.9}$	0 Broken	$7.43s$	$85.79s$	$8.21s$
	1 Broken	∞	$98.49s$	$25.64s$
	2 Broken	∞	$109.06s$	$74.22s$
λ_2	0 Broken	-1.0	-0.029	-0.66
	1 Broken	0	-0.026	-0.049
	2 Broken	0	-0.023	-0.024



Getting libff

- Install CCL
  http://soslab.ee.washington.edu/mw/index.php/Code

- Get the source code for libff
  http://soslab.ee.washington.edu/nnapp/wiki/ff.tgz

- Delete any existing ff subdirectories of the CCL root directory and copy the new source into CCL_ROOT/ff

- Call make, it should build the binaries and install them and CCL API into the appropriate locations