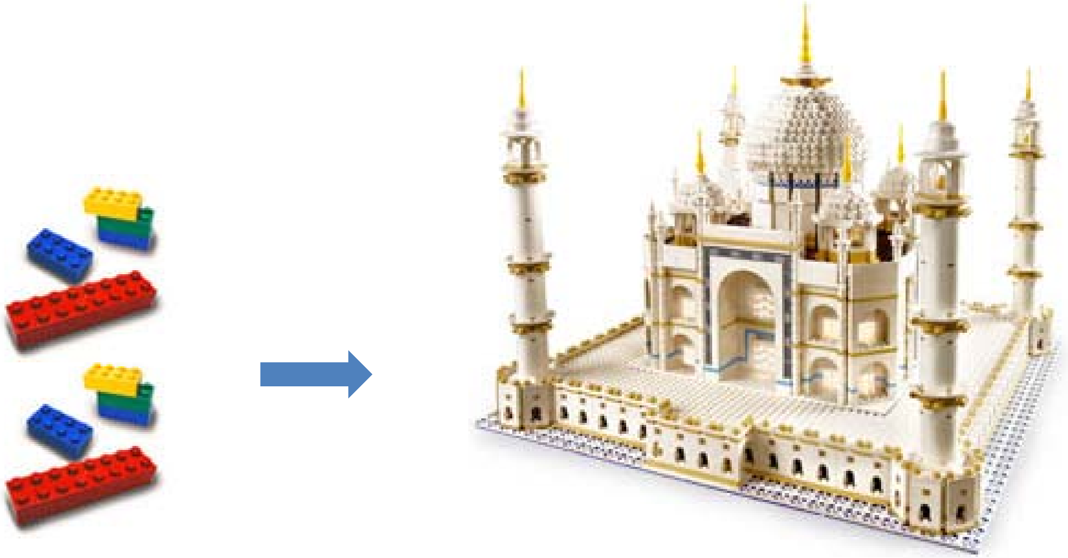
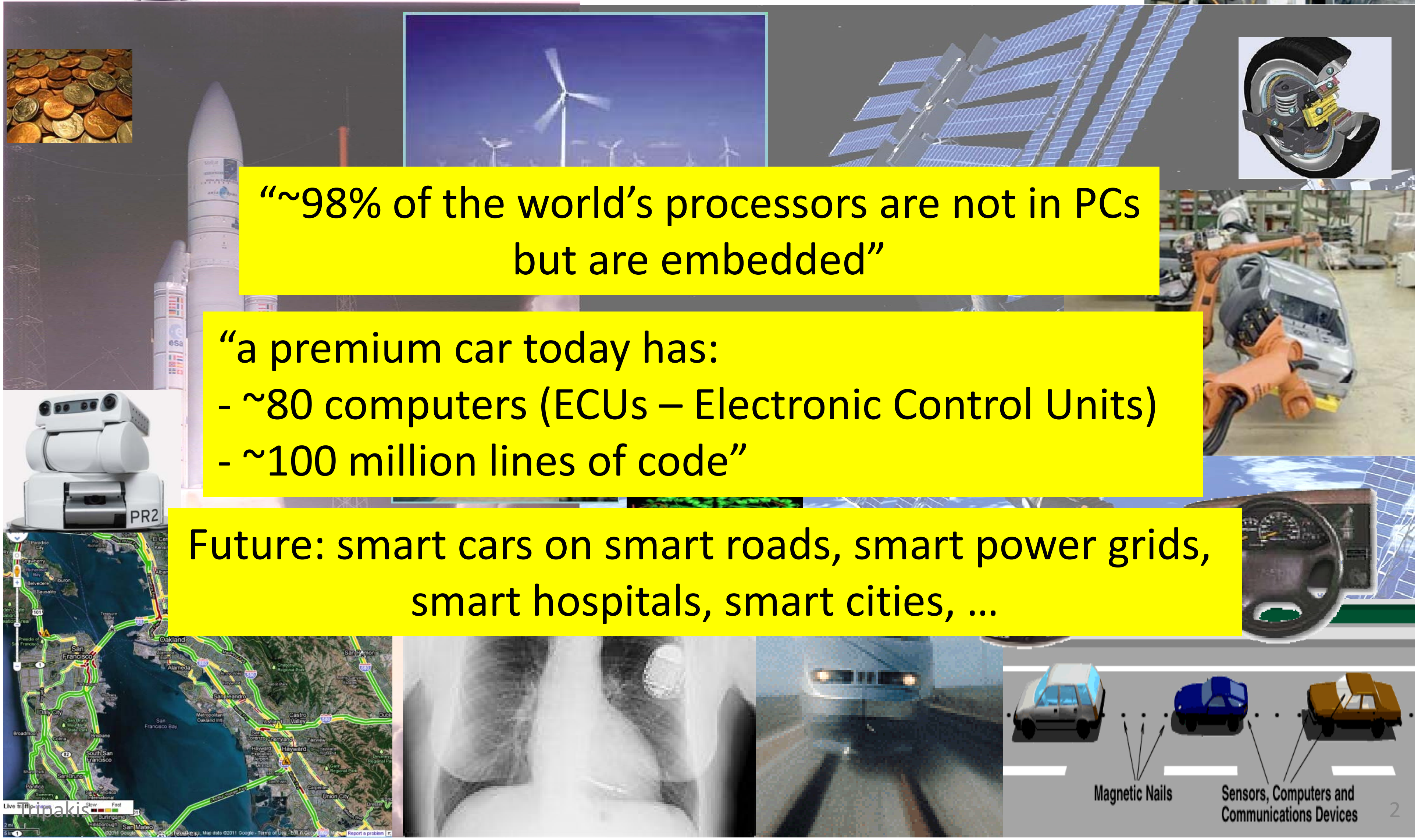
Compositionality in system design: interfaces everywhere!

Stavros Tripakis
UC Berkeley



DREAMS Seminar, Mar 2013

Computers as parts of cyber-physical systems



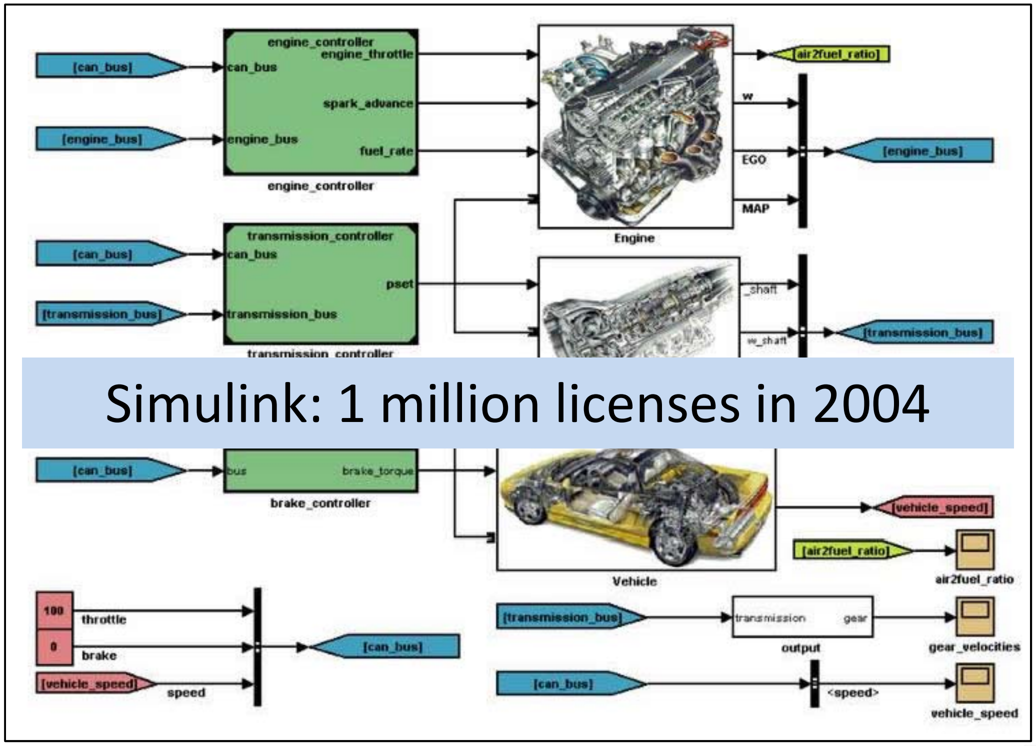
"~98% of the world's processors are not in PCs but are embedded"

"a premium car today has:
- ~80 computers (ECUs – Electronic Control Units)
- ~100 million lines of code"

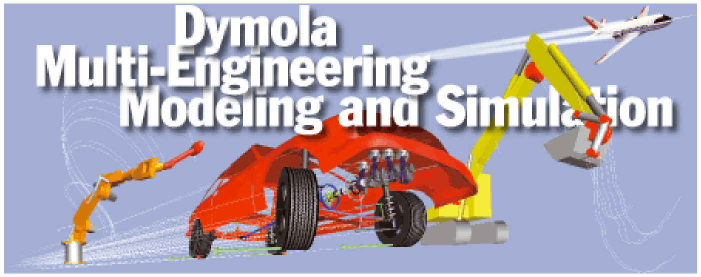
Future: smart cars on smart roads, smart power grids, smart hospitals, smart cities, …

Languages & tools for CPS

Simulink

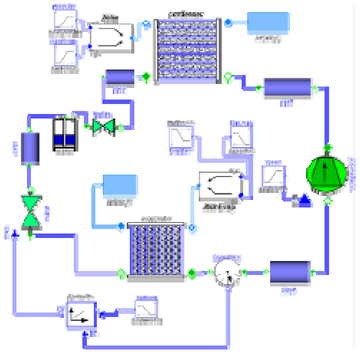


Simulink: 1 million licenses in 2004

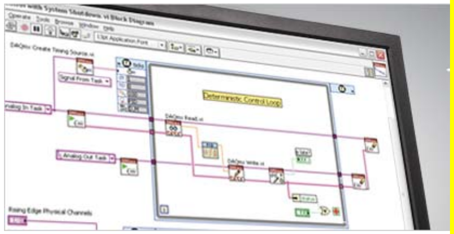


Key concepts:
- reactive behavior
- concurrency
- timing
- I/O
- …

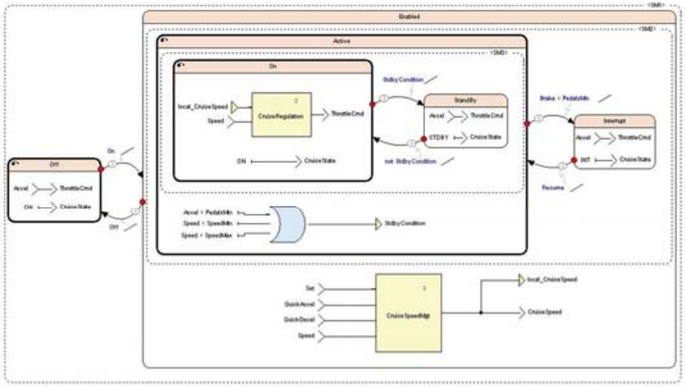
Modelica / Dymola



LabVIEW



Key capabilities:
- **simulation**
- **code generation**
- **verification**

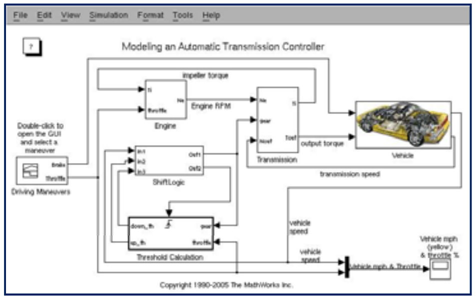


Top Level of the Cruise Control application

SCADE

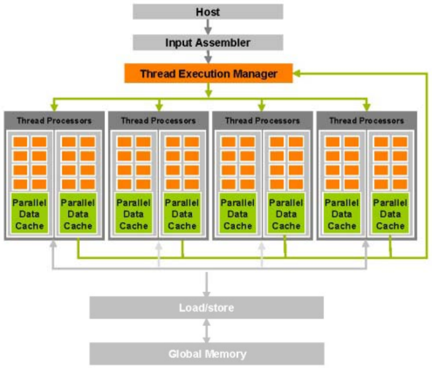
Vision

- These modeling languages of today will become the **system-programming** languages of tomorrow

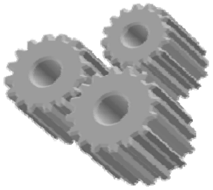


Rich languages:
concurrency, time,
robustness, reliability,
energy, security, …

system
compiler

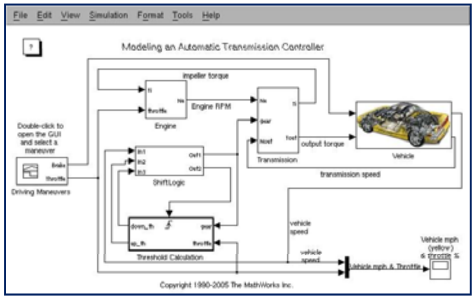


Complex
execution platforms:
networked, distributed,
multicore, …

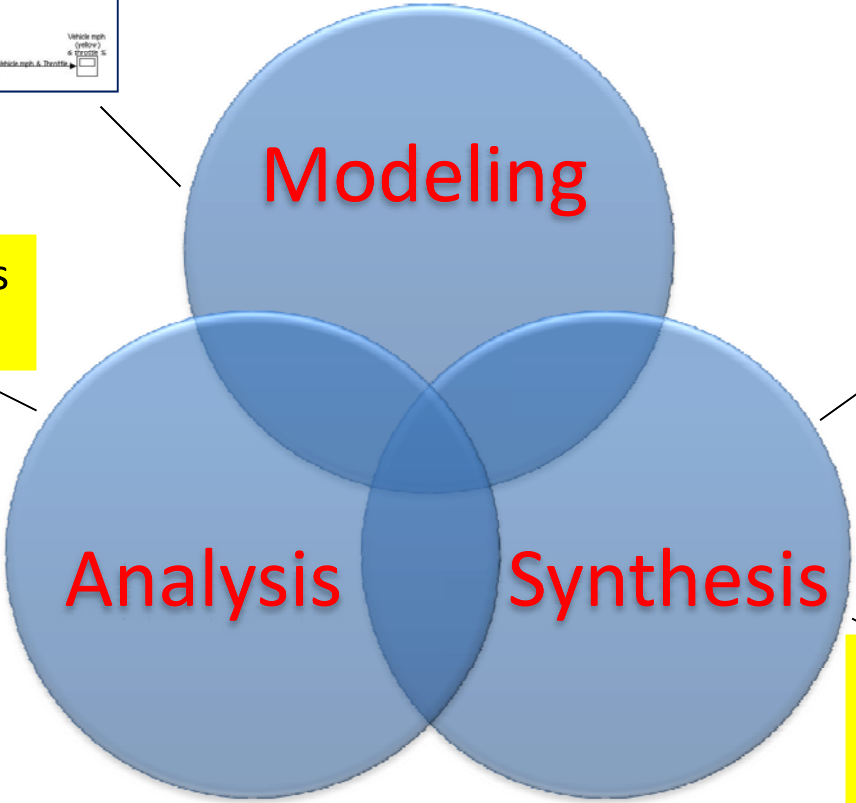


Powerful analyses:
model-checking, WCET analysis, schedulability,
performance analysis, reliability analysis, …

Model-Based Design



How to describe what we want?

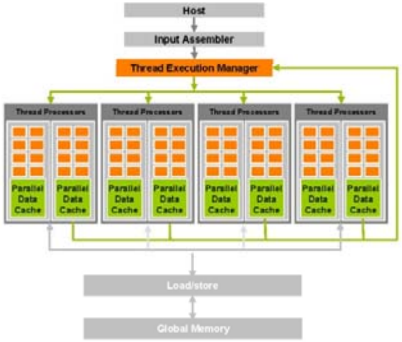


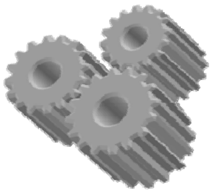
Modeling

Analysis

Synthesis

How to be sure that this is what we want?

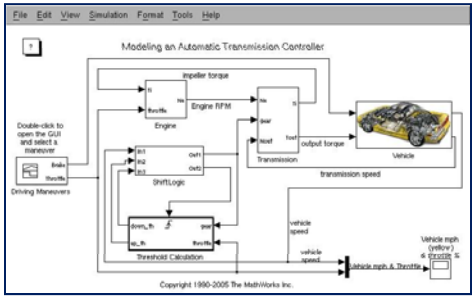




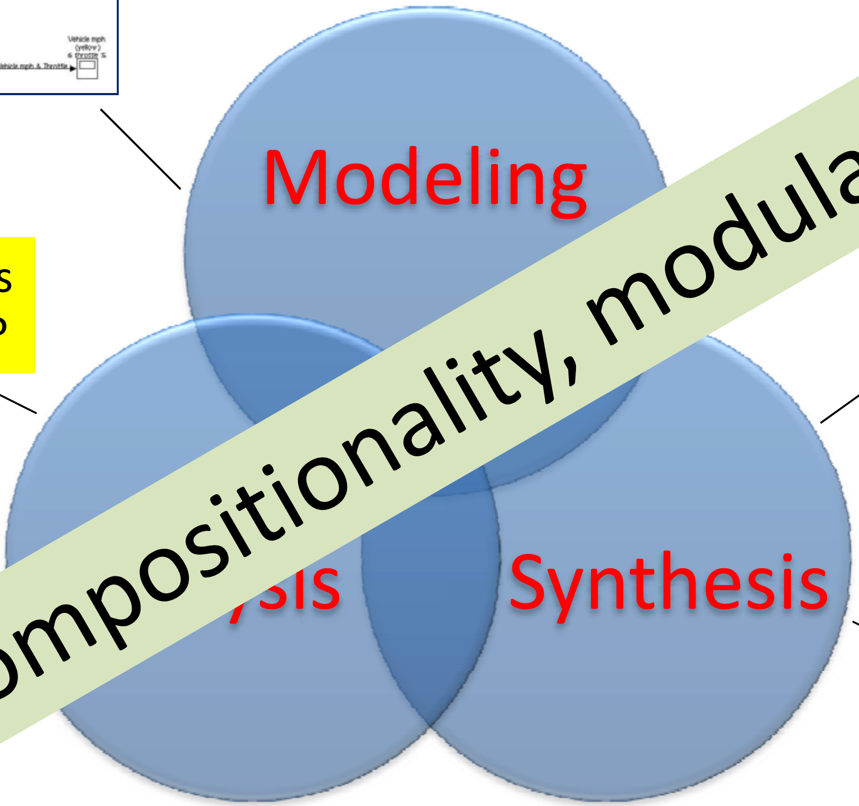
How to build it?
Automatically
Correct-by-construction

Compositional Model-Based Design

Industry: “*system integration is key*”

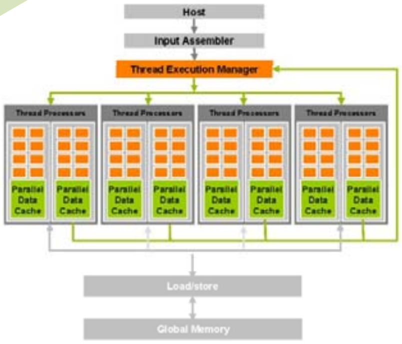


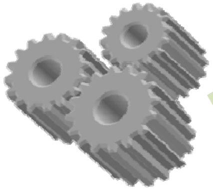
How to describe systems modularly?



Compositionality, modularity

How to check properties in a compositional way?

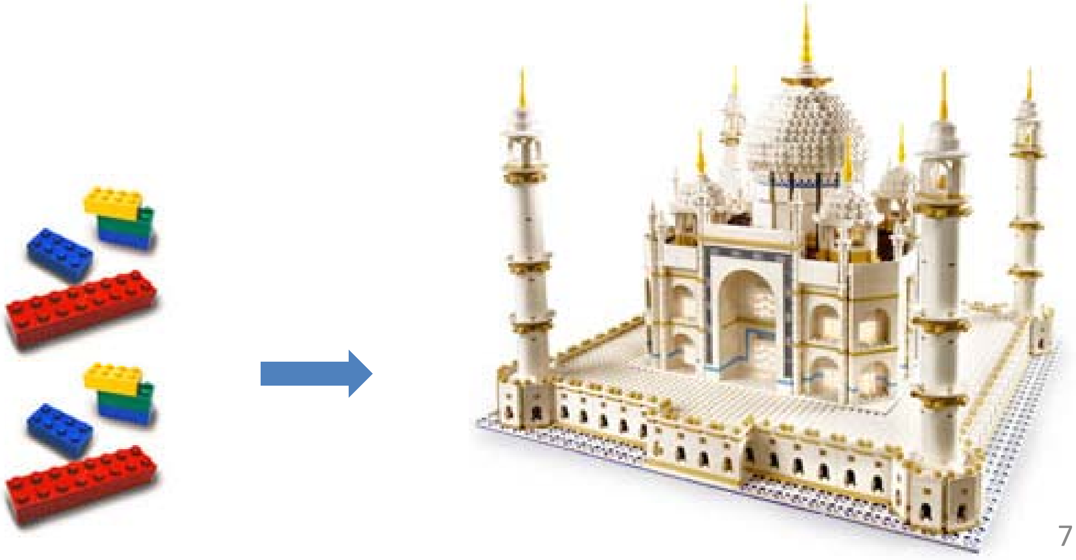




How to synthesize parts of the system independently?

This talk: interfaces

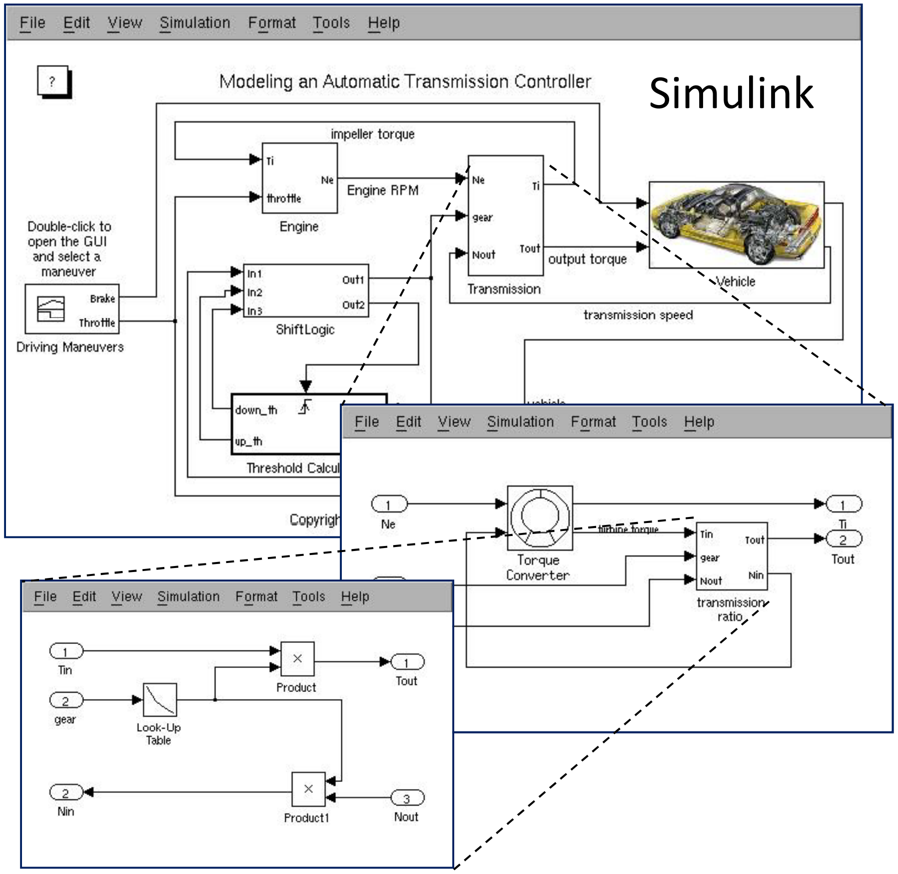
- Interface synthesis
- Interface theories

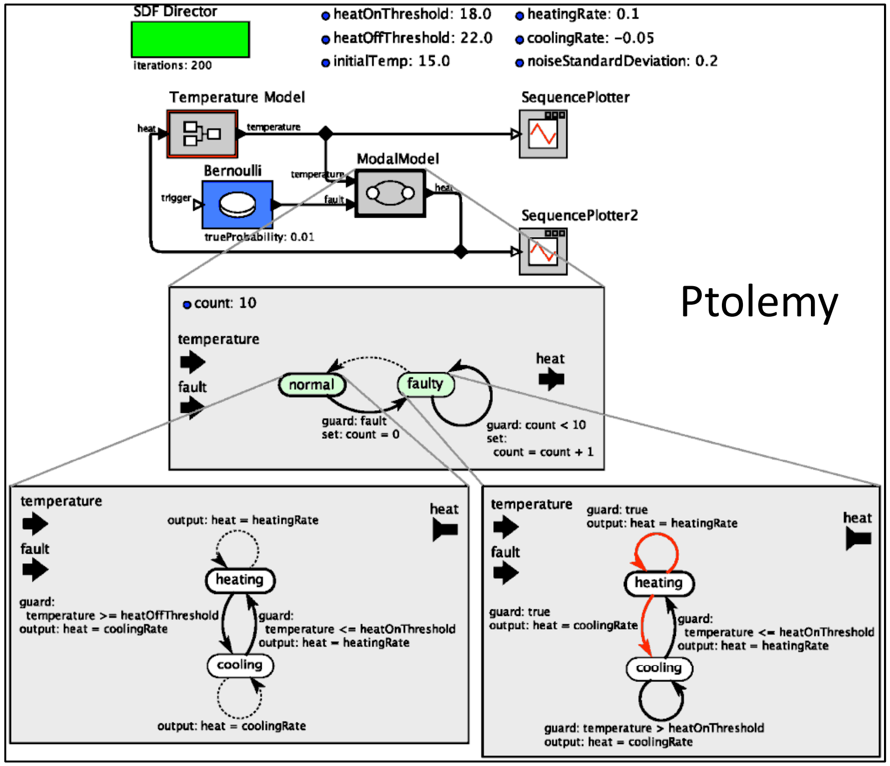


This talk

- **Interface synthesis**
  - joint with R. Lublinerman, C. Szegedy, E. Lee, M. Geilen, B. Rodiers, D. Bui
- Interface theories

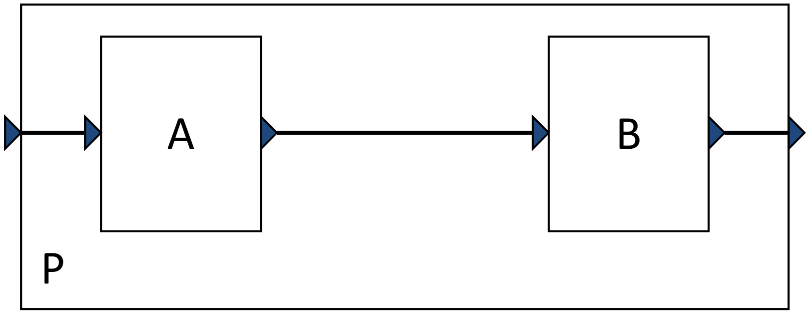
Hierarchy



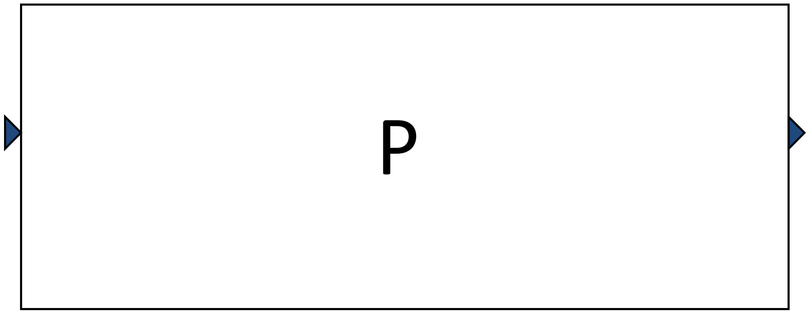


model = tree of sub-models

Hierarchy in block diagrams



Hierarchy in block diagrams

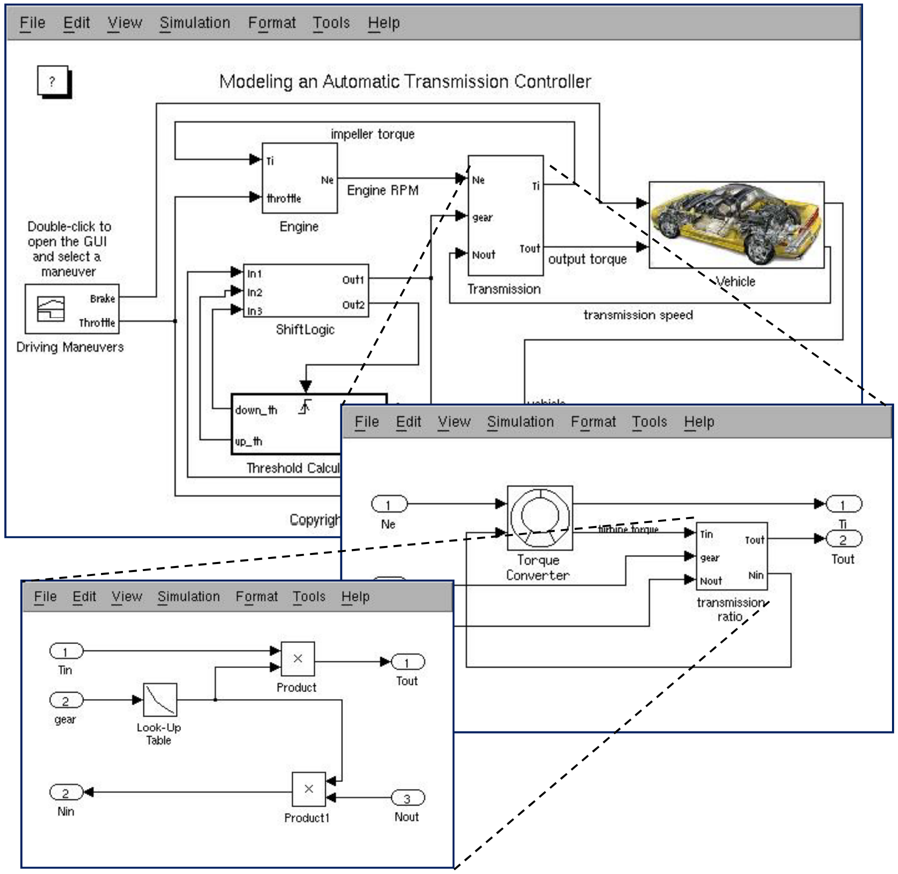


Modularization:
hide details, master complexity

Hierarchy benefits

Total number of blocks: ~100

Max. number at any level: ~6



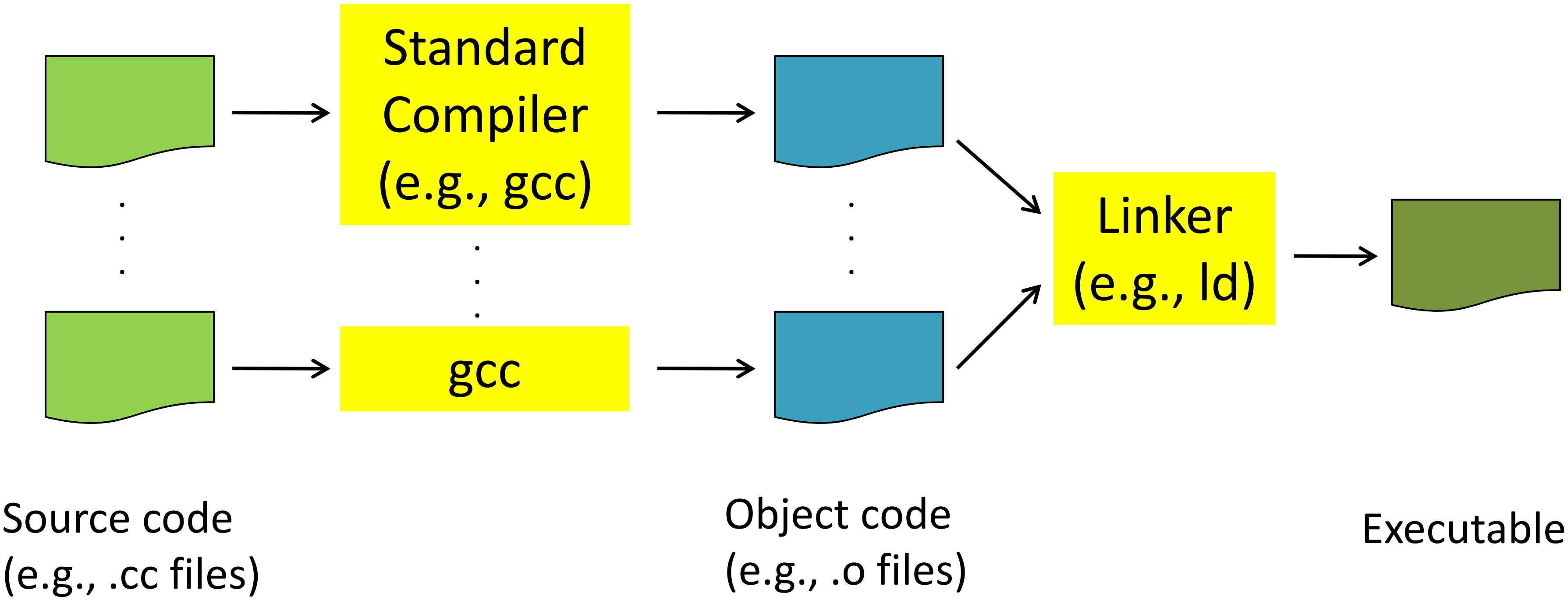
model = tree of sub-models

Can we exploit this hierarchy beyond syntax?

- Can we reuse a submodel?
- Can we treat it as a “black box”?
- Can we build model libraries?

Surprisingly, the answer is often “no” even in state-of-the-art tools

Modular compilation



Standard Compiler (e.g., gcc)

gcc

Linker (e.g., ld)

Source code
(e.g., .cc files)

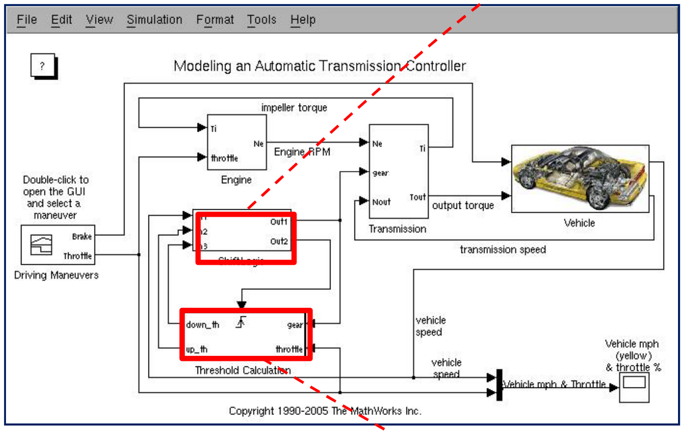
Object code
(e.g., .o files)

Executable

Enables incremental compilation, IP protection, …, and libraries!

Modular code generation

Can we do the same for system-design languages?



automatic code generation

```
initialize state;
while (true) do
  await clock tick;
  read inputs;
  compute;
  write outputs;
  update state;
end while;
```

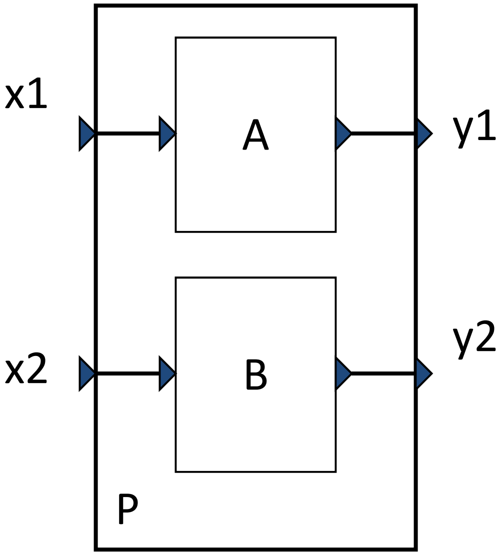
Code for block A

```
initialize state;
while (true) do
  await clock tick;
  read inputs;
  compute;
  write outputs;
  update state;
end while;
```

Code for block B

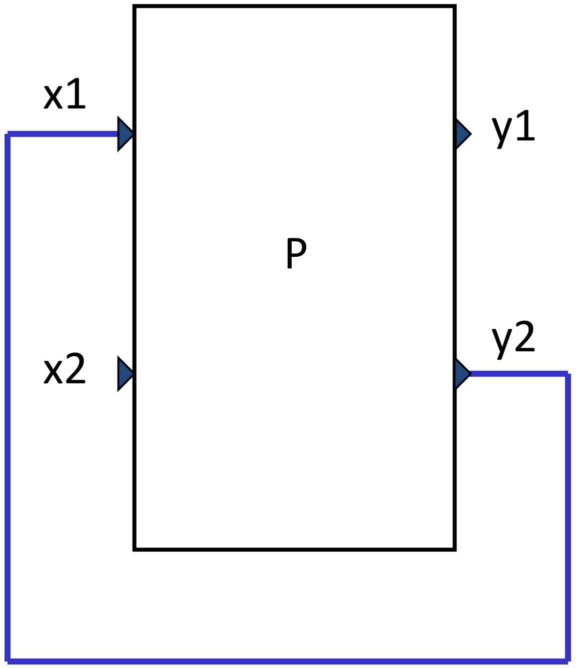
Code
(C, C++, Java, …)

Standard code generation: “monolithic”



```
P.fire(x1, x2) returns (y1, y2)
{
     y1 := A.fire( x1 );
     y2 := B.fire( x2 );

     return (y1, y2);
}
```

Problem with “monolithic” code



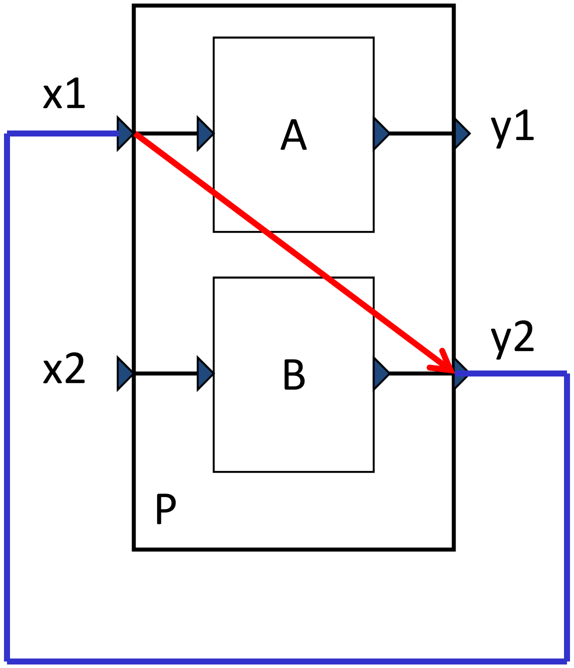
```
P.fire(x1, x2) returns (y1, y2);
```

Problem with “monolithic” code

False I/O dependencies

=>

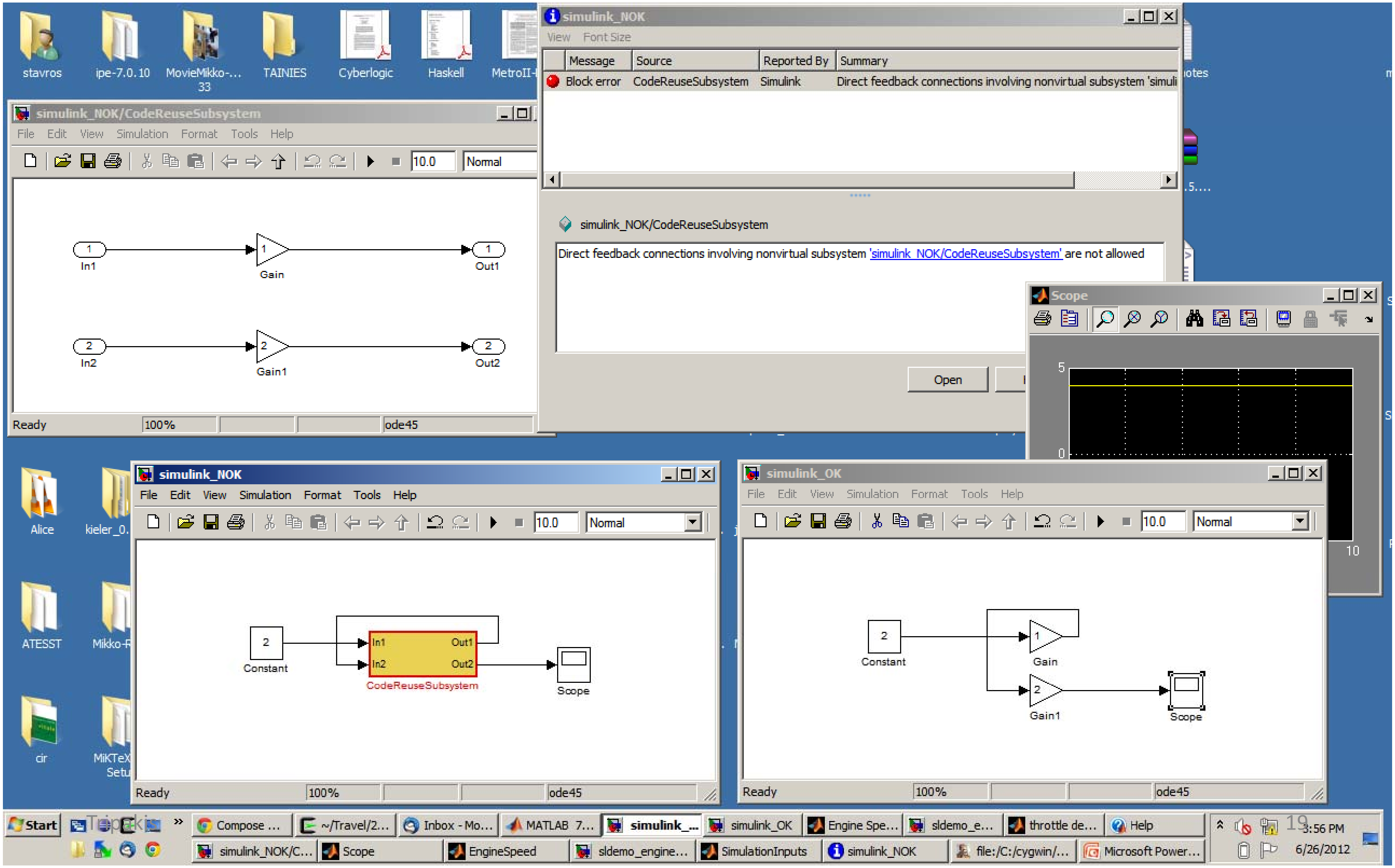
code not usable in some contexts



Parallel composition of functions is not a function!

```
P.fire(x1, x2) returns (y1, y2);
```

Brief Simulink demo



simulink_NOK

Message	Source	Reported By	Summary
Block error	CodeReuseSubsystem	Simulink	Direct feedback connections involving nonvirtual subsystem 'simuli

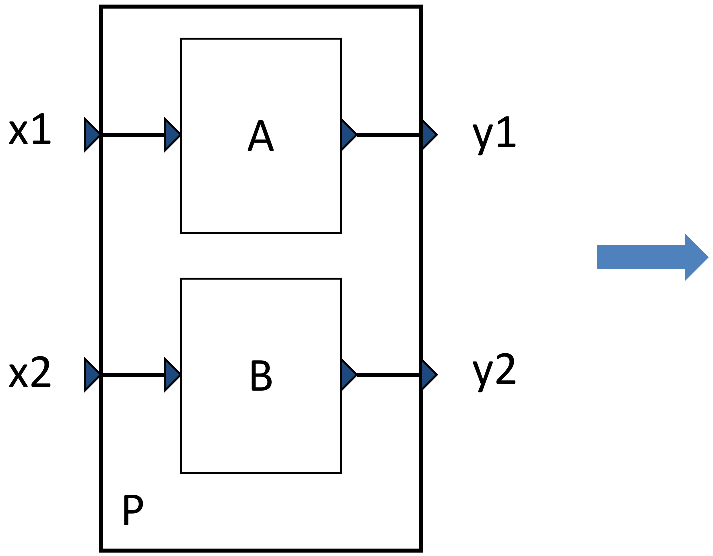
simulink_NOK/CodeReuseSubsystem

Direct feedback connections involving nonvirtual subsystem 'simulink_NOK/CodeReuseSubsystem' are not allowed

[DATE’08, POPL’09]

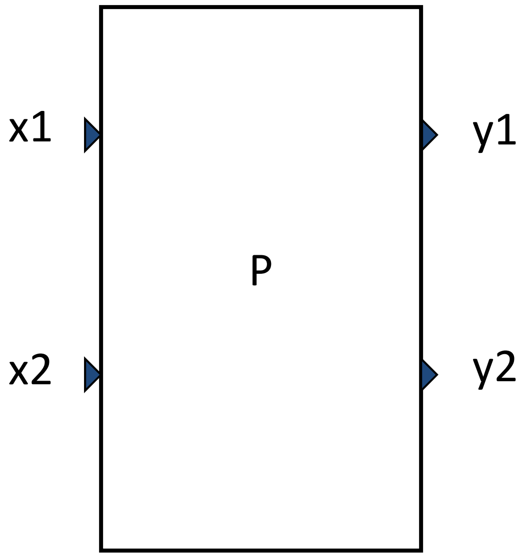
Solution: non-monolithic code

A Mealy machine with multiple output functions



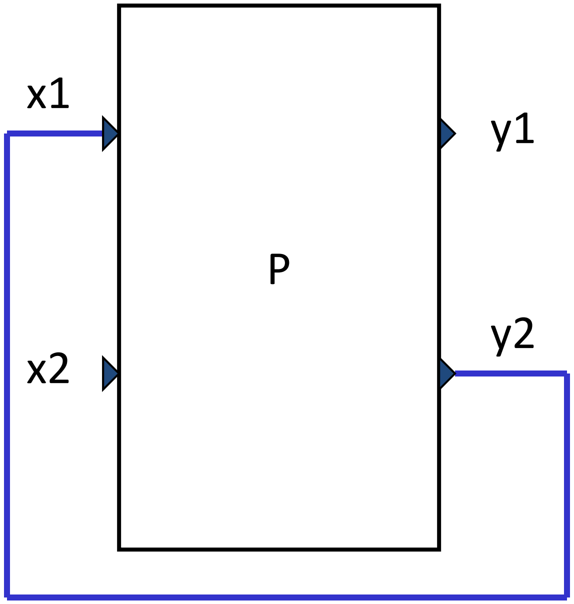
```
P.fire1( x1 ) returns y1 {
   return A.fire( x1 );
}


P.fire2( x2 ) returns y2 {
   return B.fire( x2 );
}
```

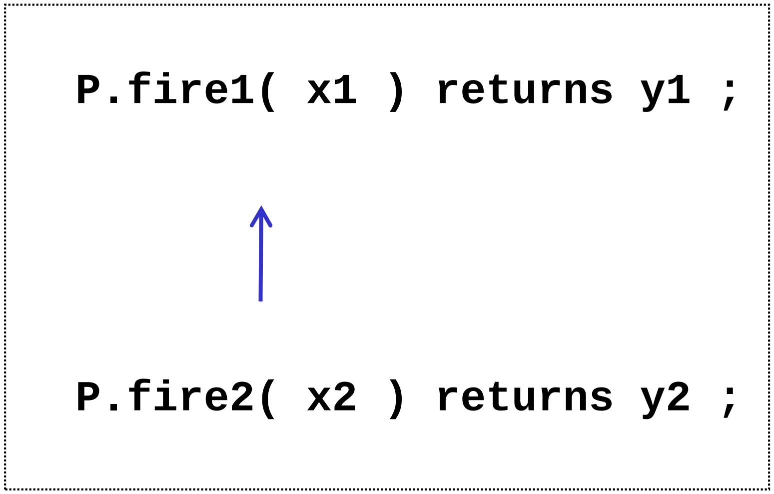
Non-monolithic interface



```
P.fire1( x1 ) returns y1 ;


P.fire2( x2 ) returns y2 ;
```

Non-monolithic interface

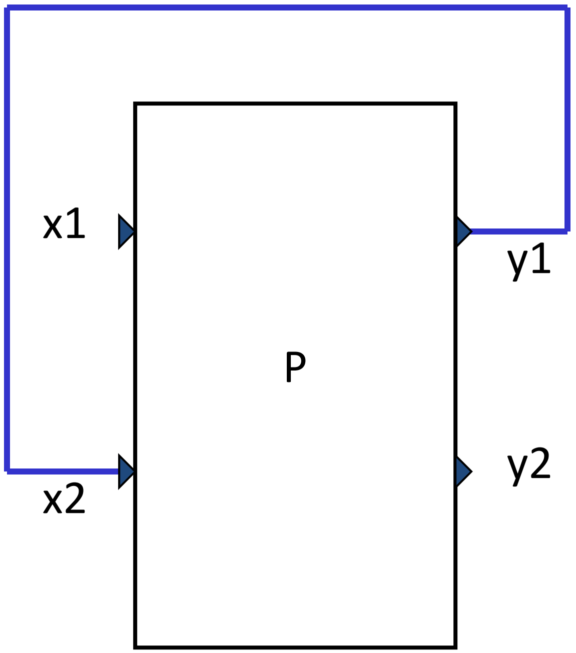




```
P.fire1( x1 ) returns y1 ;

P.fire2( x2 ) returns y2 ;
```

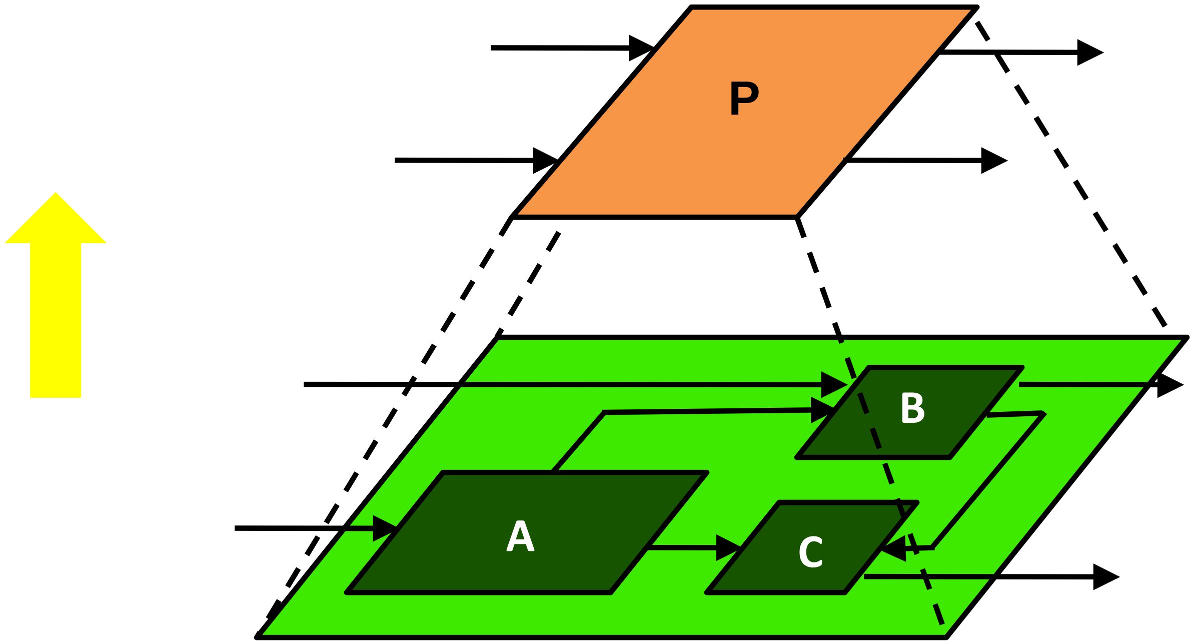
Non-monolithic interface

interface does not restrict usage



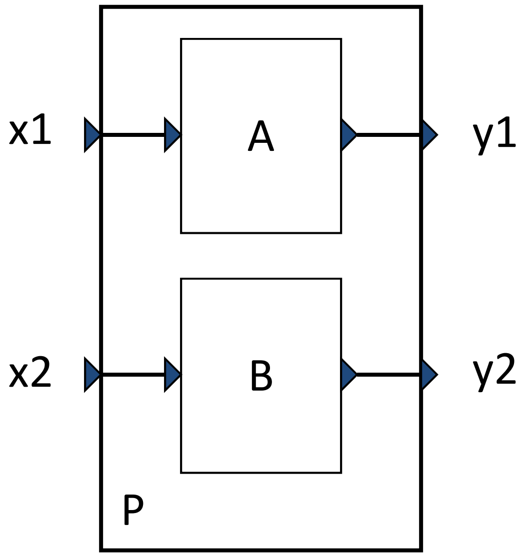
```
P.fire1( x1 ) returns y1 ;

        ↓

P.fire2( x2 ) returns y2 ;
```

Bottom-up interface synthesis



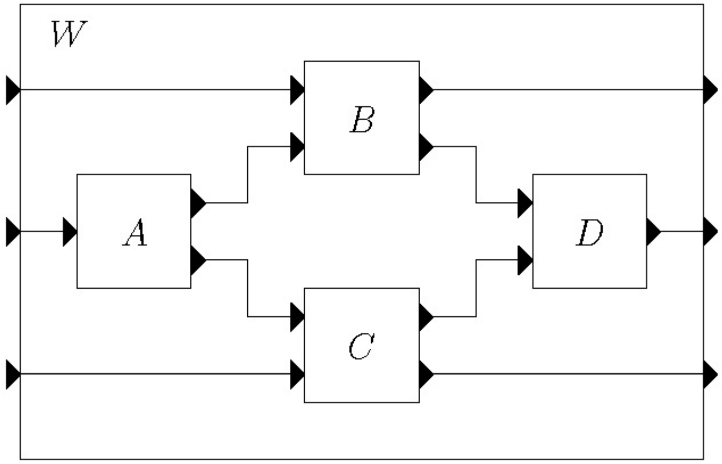
Given interfaces for sub-blocks A, B, C, compute interface for composite block P.

Simple non-monolithic interface

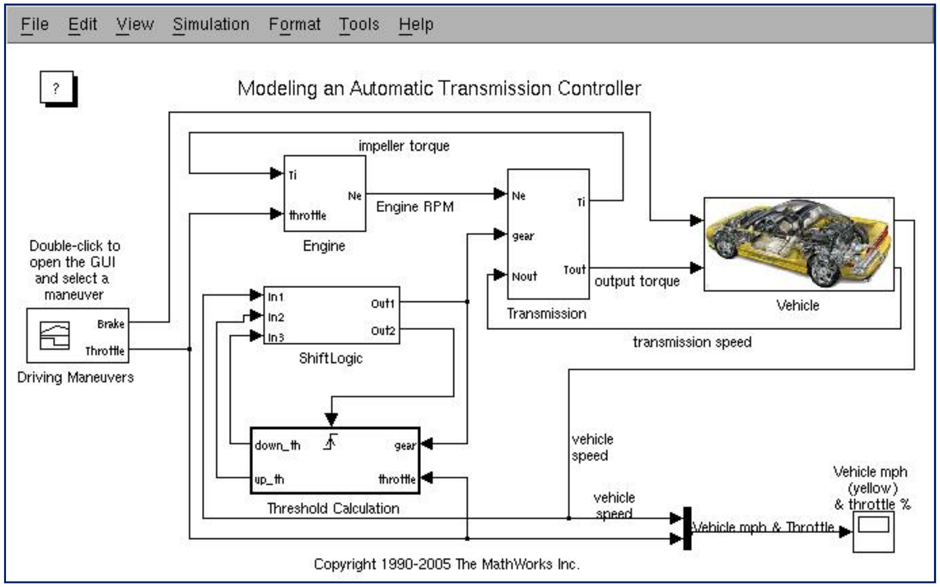


```
P.fire1( x1 ) returns y1 ;


P.fire2( x2 ) returns y2 ;
```

What about more complex diagrams?

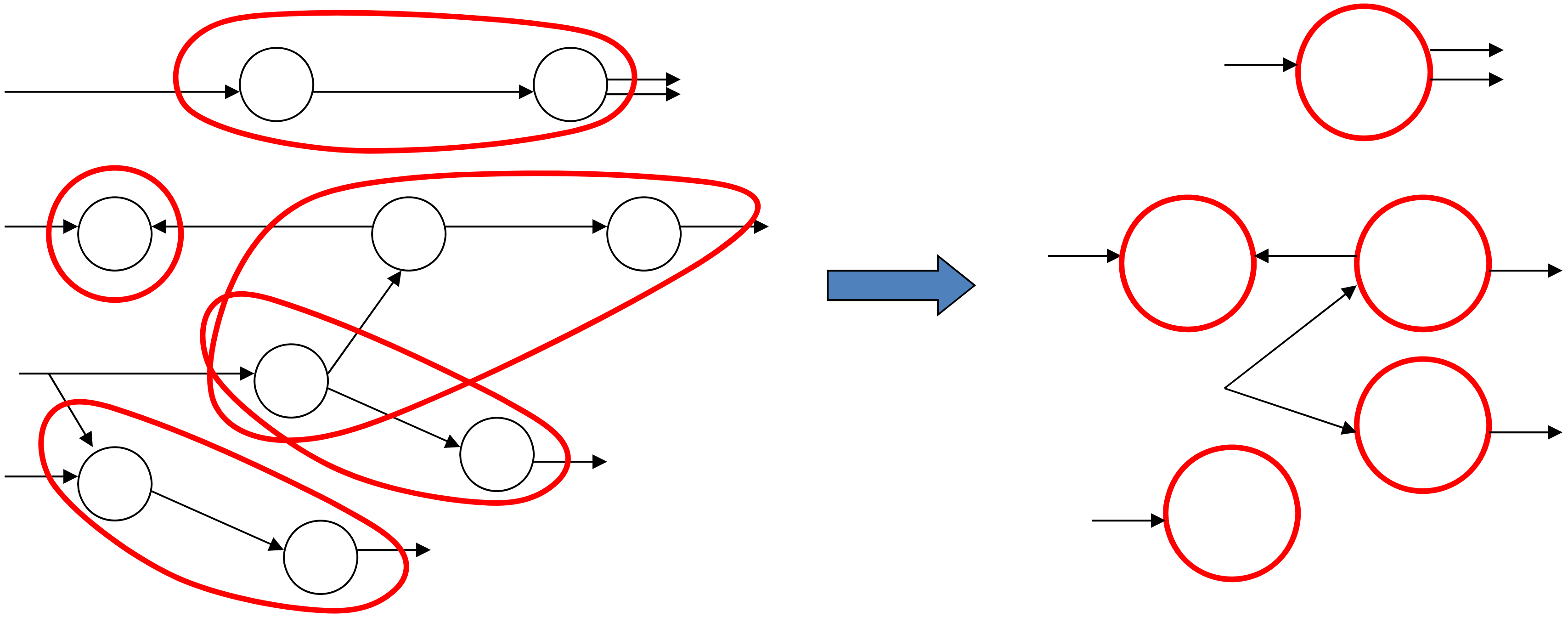


what about this?



or this?

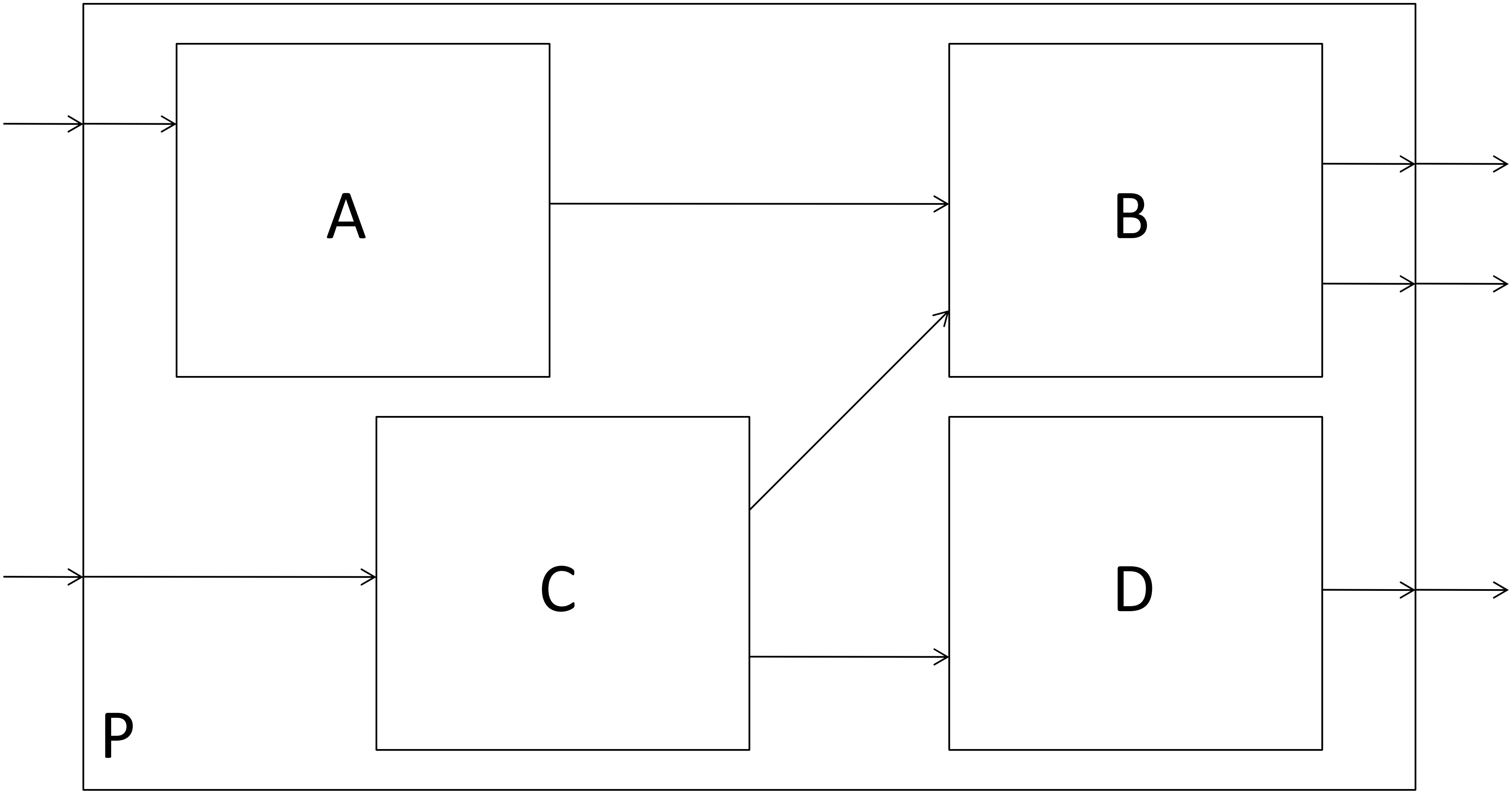
Interface synthesis for block diagrams = graph clustering



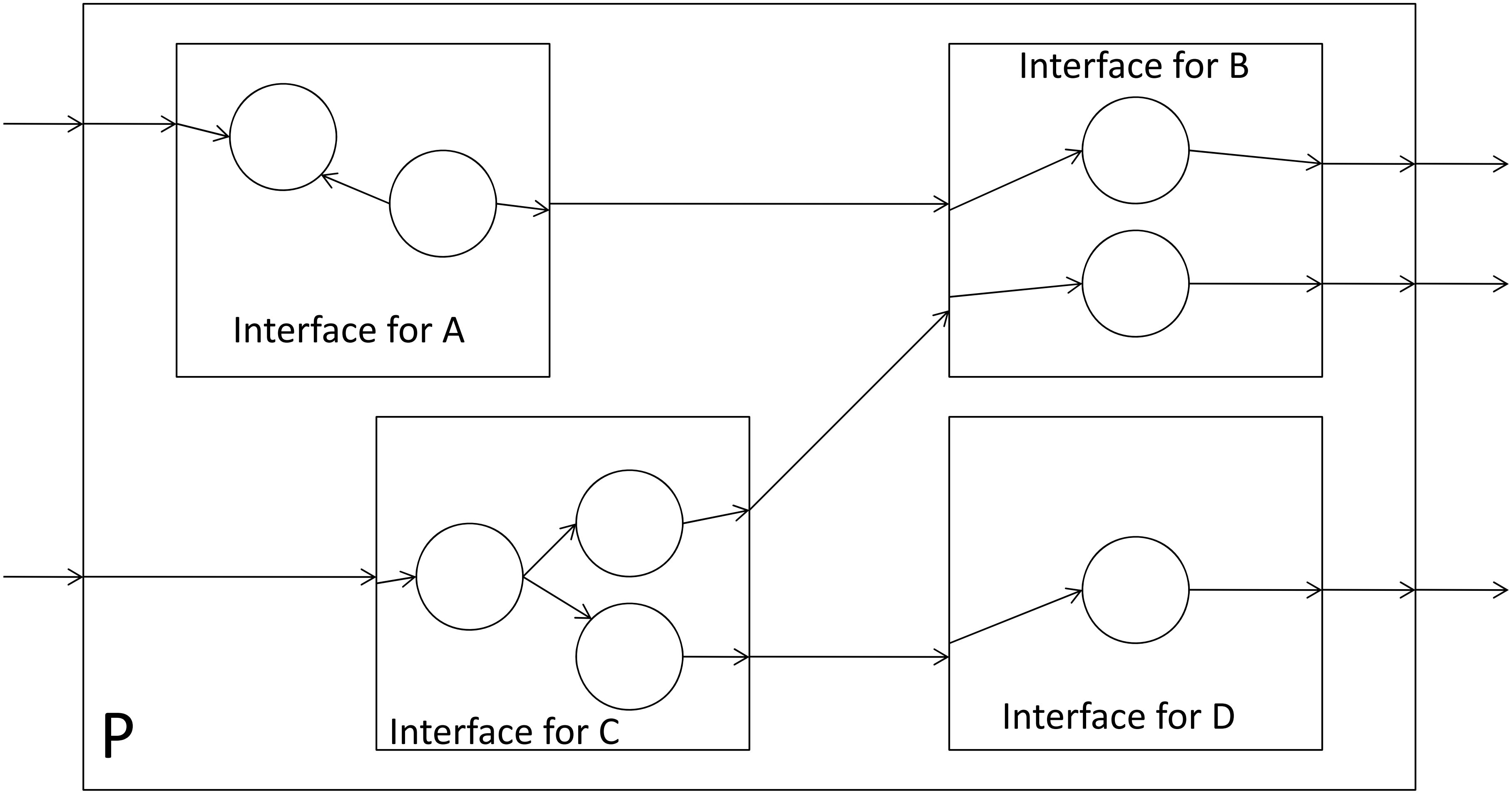
block diagram

interface

How it’s done

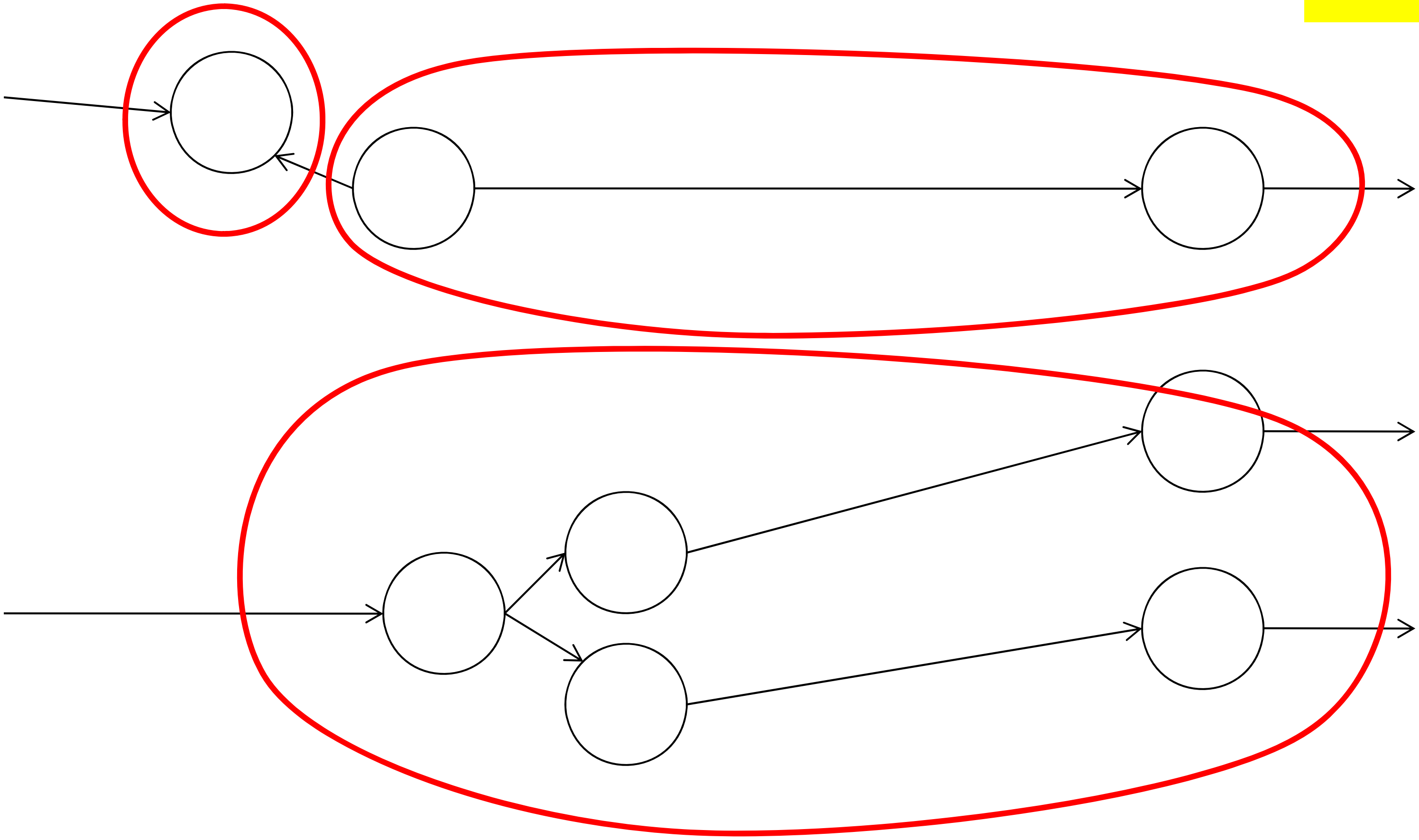


How it’s done

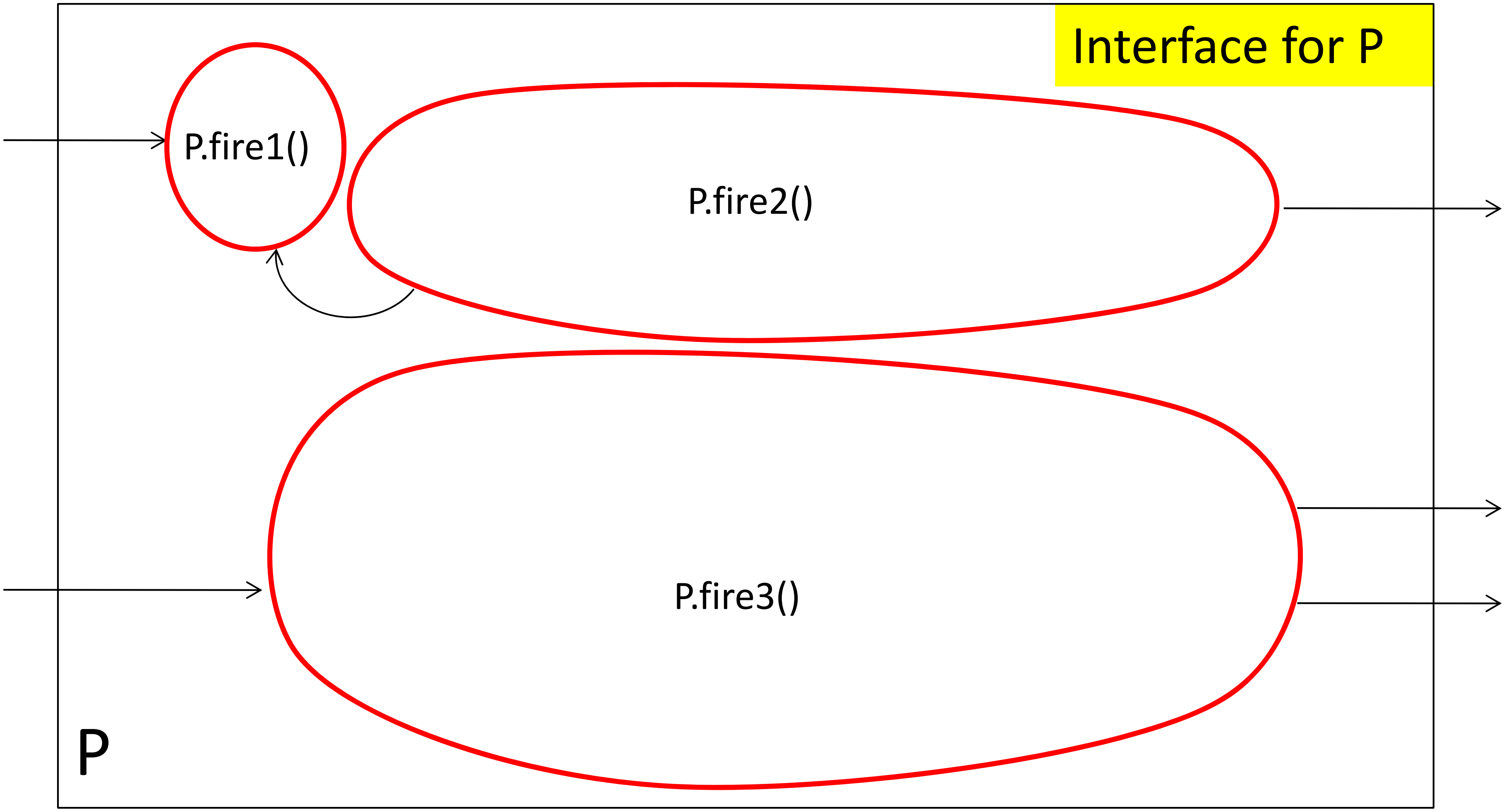

Interface for A
Interface for B
Interface for C
Interface for D
P

How it's done

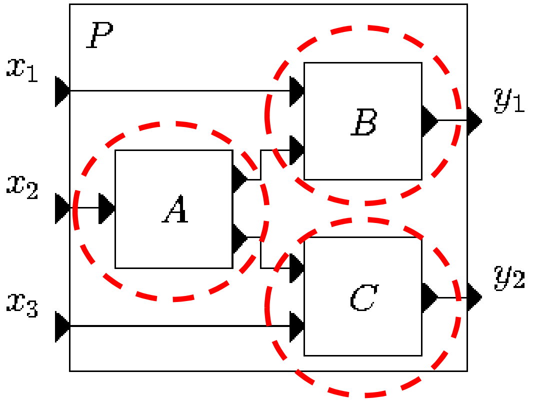
clustering

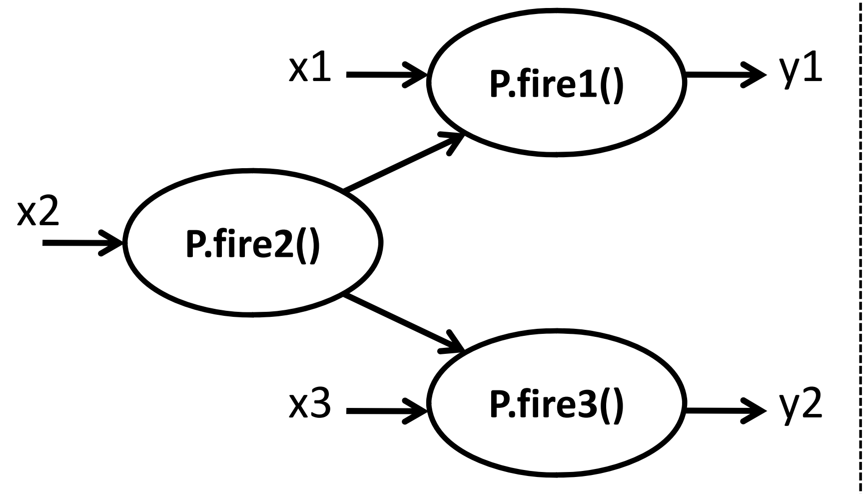


How it's done

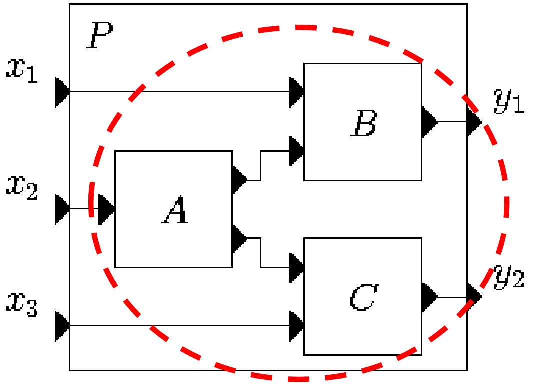


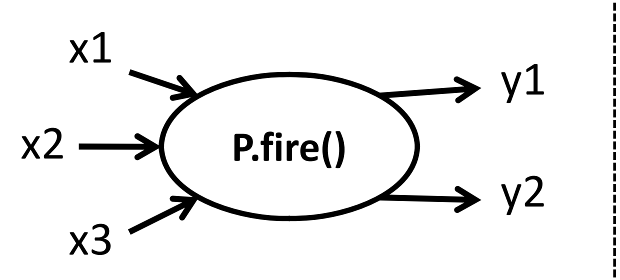
Different clusterings => different interfaces





A **non-monolithic** interface for P





A **monolithic** interface for P

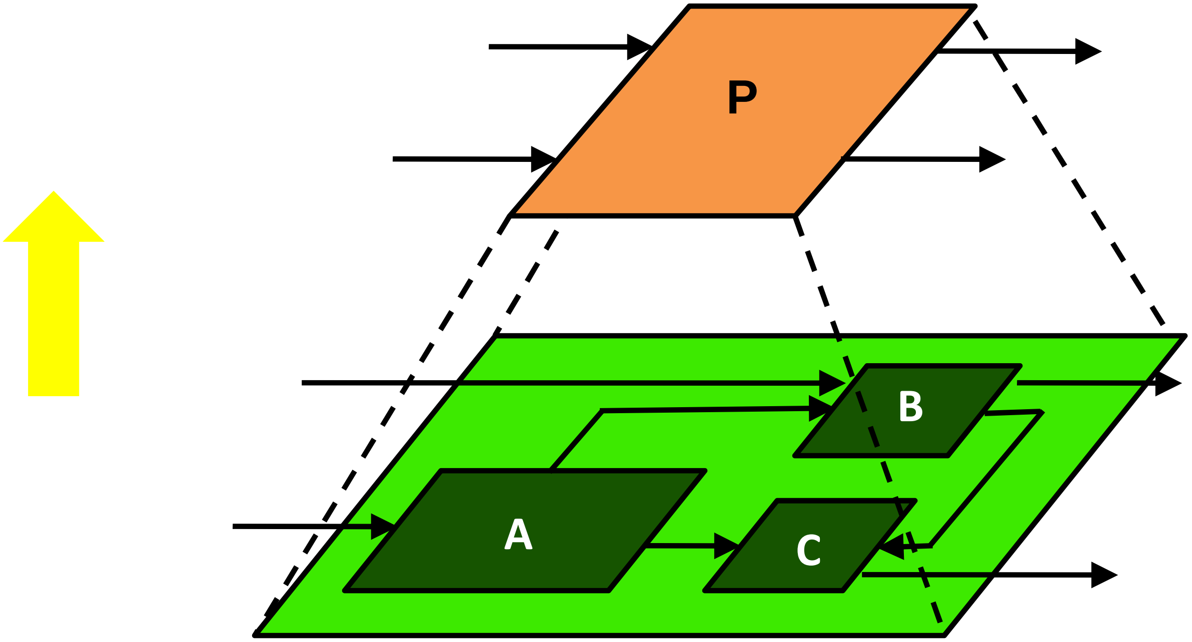
trade-off: interface size vs. reusability

Different clustering algorithms = different tradeoffs

Clustering method	Complexity	Achieves maximal reusability?	Achieves minimal interf. size?	Modularity bound?	Achieves minimal code size?
“step-get”	Polynomial	No	Almost	<=2 functions	Yes
“dynamic”	Polynomial	Yes	Yes	<=N+1 functions*	No
“disjoint”	NP-complete	Yes	Yes	?	Yes
“greedy”	Polynomial	Yes	No	?	Yes

* N = number of block outputs

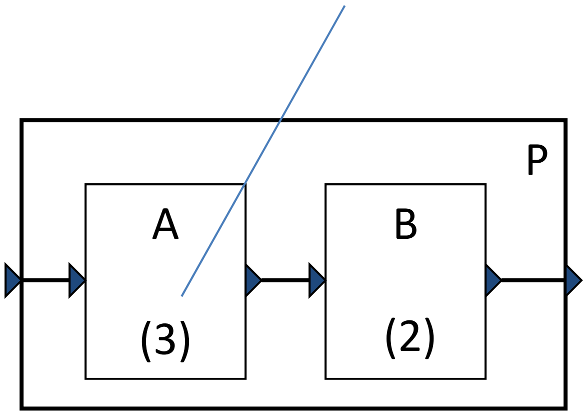
Bottom-up interface synthesis = automatic abstraction



Efficient + provably optimal

[RTAS'08]

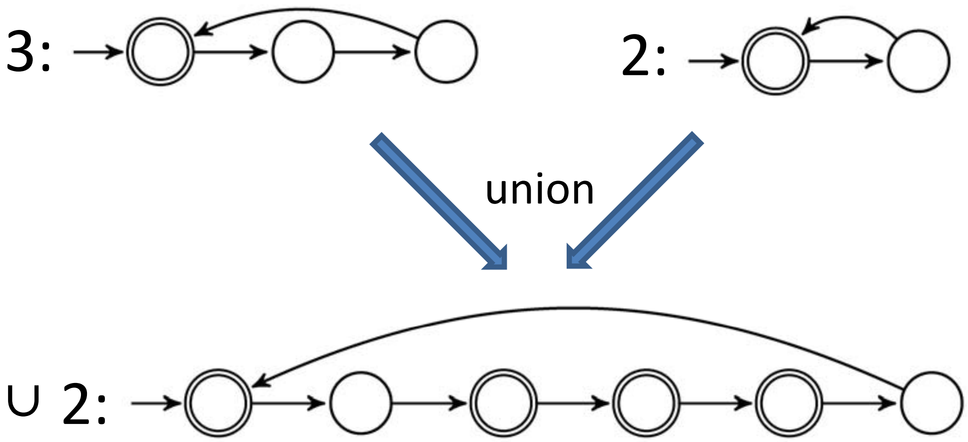
Interfaces for multirate models



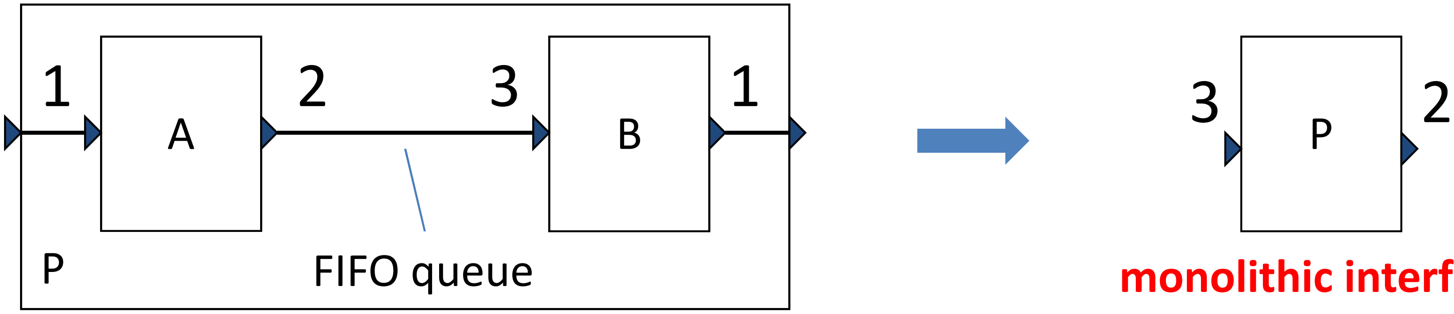
Block A fires every 3 rounds

When does P need to fire?
GCD(3,2) = 1 : over-approximation
too conservative

Interface enriched with
deterministic unary automata



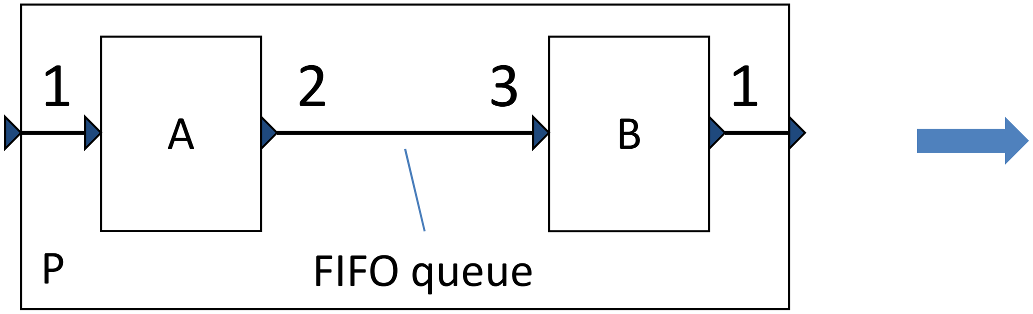
Interfaces for dataflow models

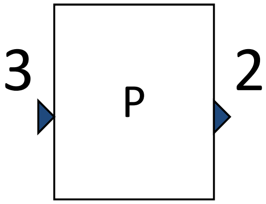


monolithic interface

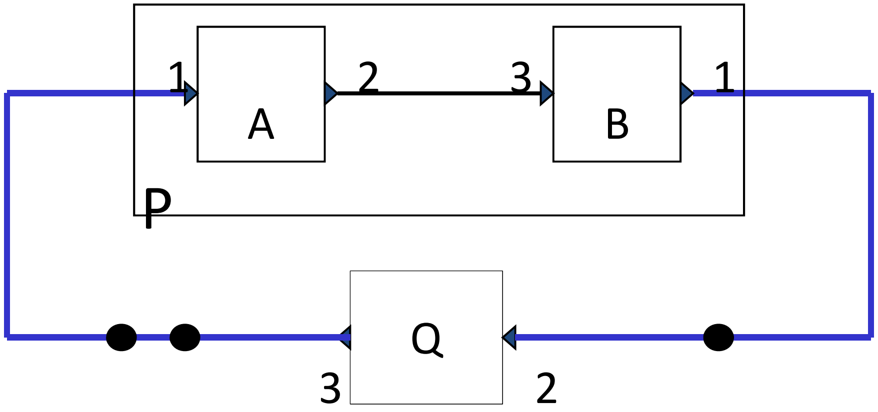
[ACM TECS 2013]

Interfaces for dataflow models

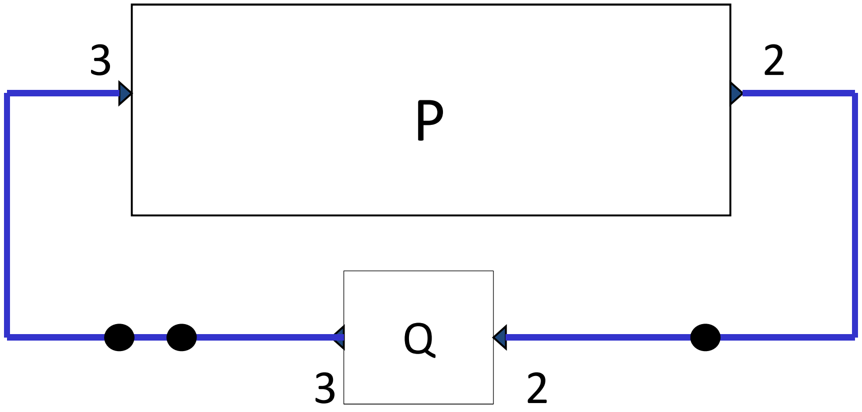




monolithic interface



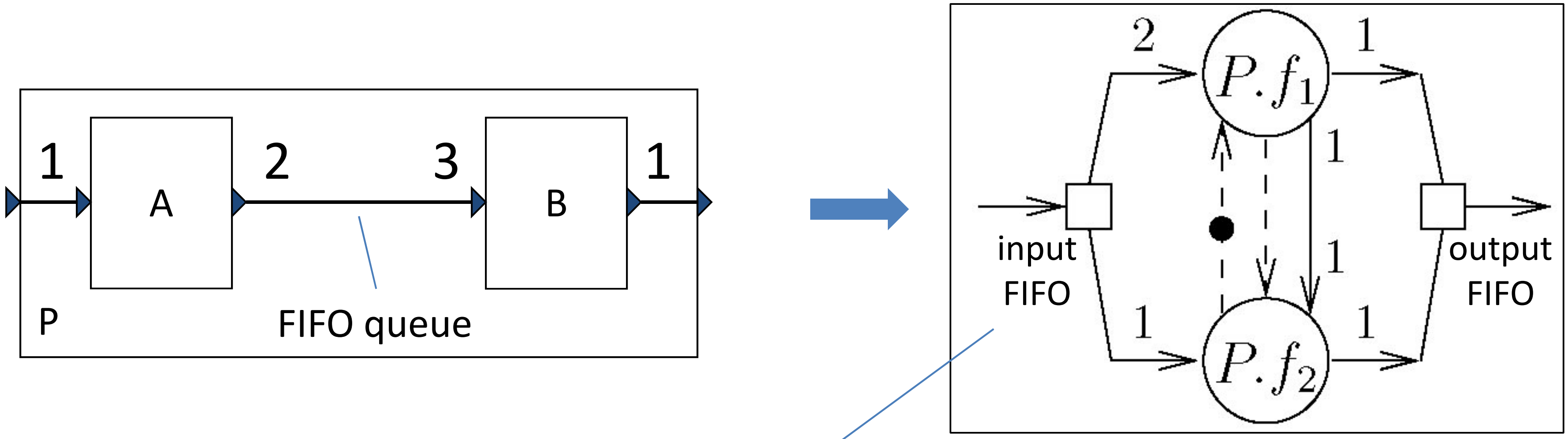
original model does not deadlock



monolithic interface
=> deadlocks!

[ACM TECS 2013]

Interfaces for dataflow models



SDF++ :

- shared FIFOs
- but deterministic!

Non-monolithic interface for P
(generated automatically)

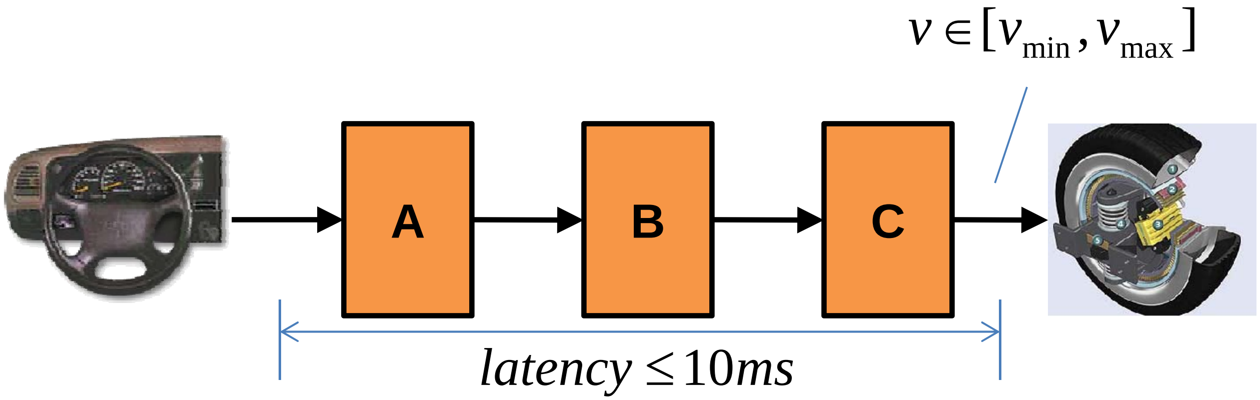
This talk

- Interface synthesis

- **Interface theories**
  - joint with B. Lickly, T. Henzinger, E. Lee, M. Geilen, M. Wiggers, C. Stergiou, M. Broy

Incremental design

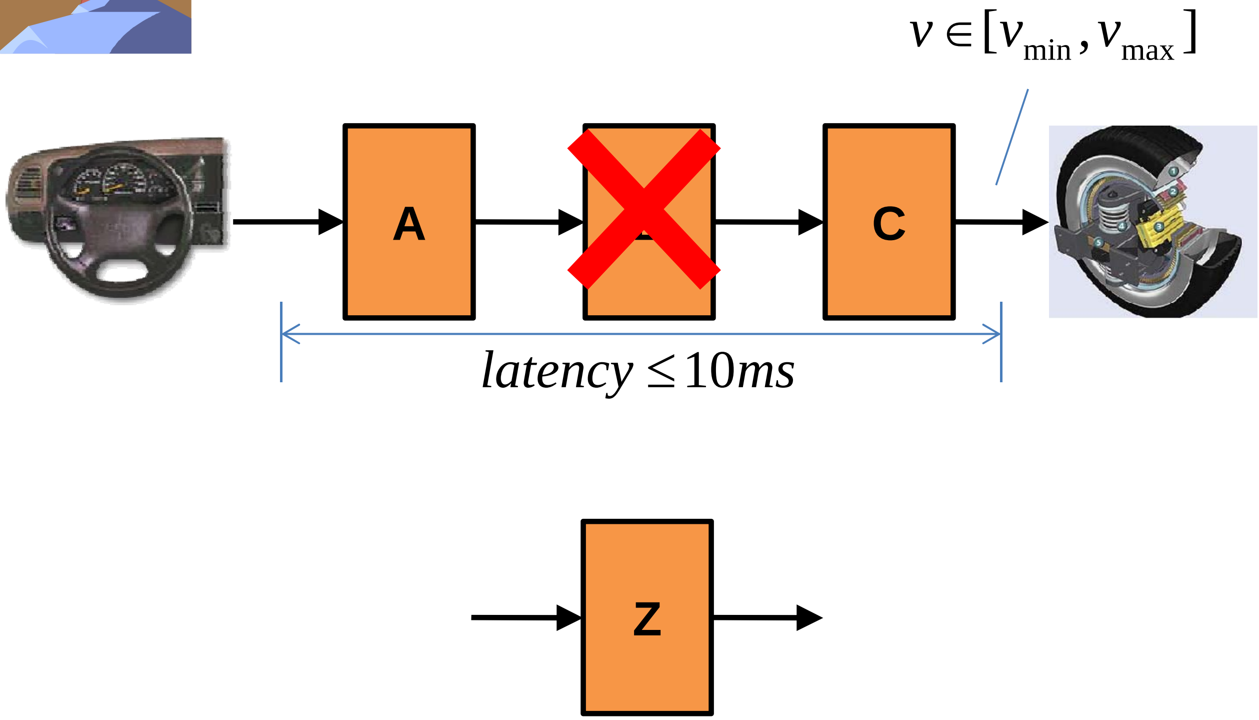
A “steer-by-wire” system:



$v \in [v_{\min}, v_{\max}]$

$latency \leq 10ms$

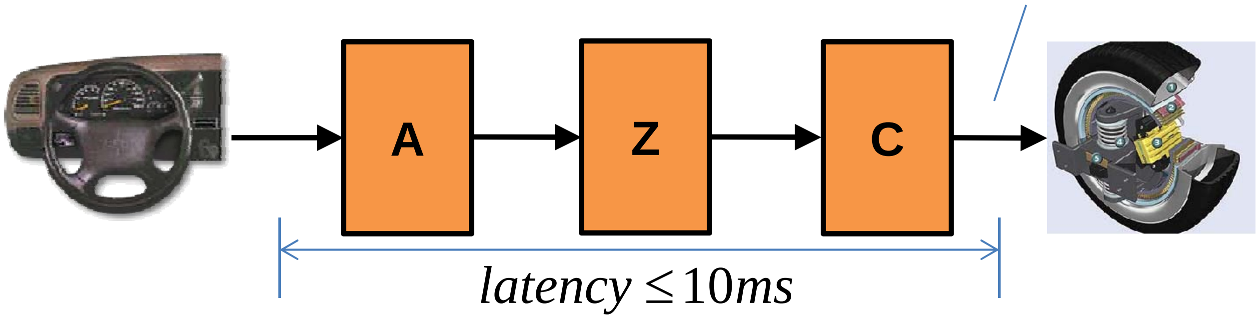
Incremental design



Incremental design

How to ensure properties are preserved?

$v \in [v_{\min}, v_{\max}]$



$latency \leq 10ms$

Interface theories [Alfaro, Henzinger, et al.]

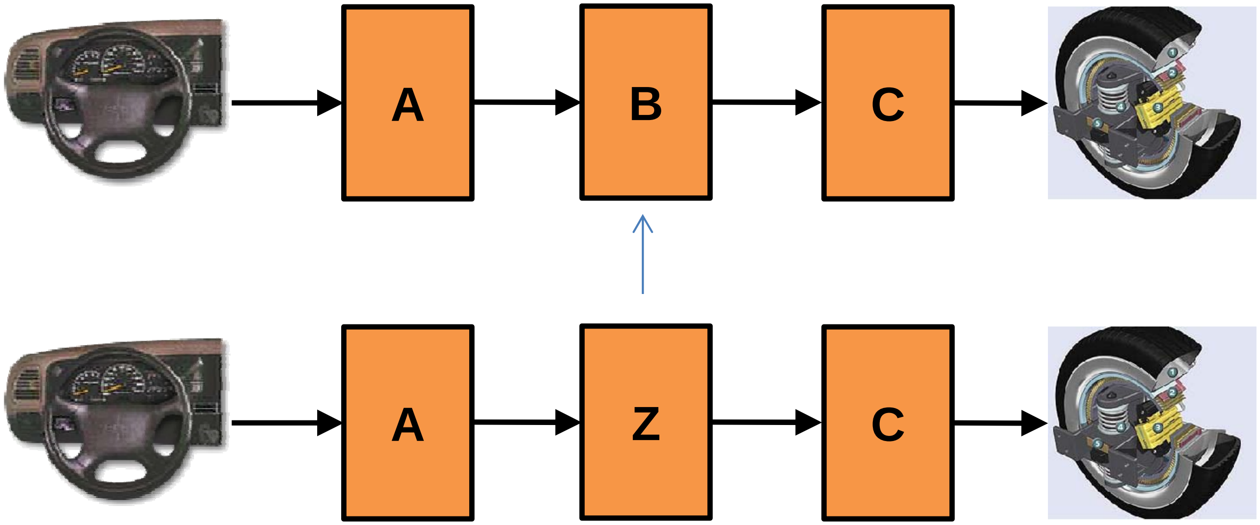
- Interface = component abstraction
- Interface composition: A • B = C
- Interface refinement: A’ ≤ A

- Theorems:

(1) If A’ ≤ A and A satisfies P then A’ satisfies P.

(2) If A’ ≤ A and B’ ≤ B, then A’ • B’ ≤ A • B.

Substitutability



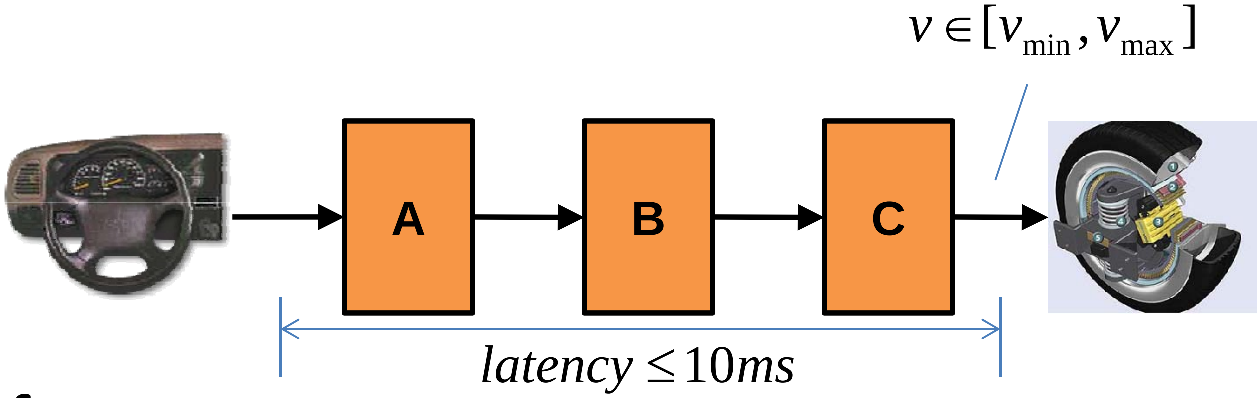
(1) If $A' \leq A$ and A satisfies P then A' satisfies P.

(2) If $A' \leq A$ and $B' \leq B$, then $A' \bullet B' \leq A \bullet B$.

$Z \leq B$ and (1) and (2) => substitutability

Which interface theories for CPS?

- Synchronous relational interfaces
  - Functional properties (correctness)



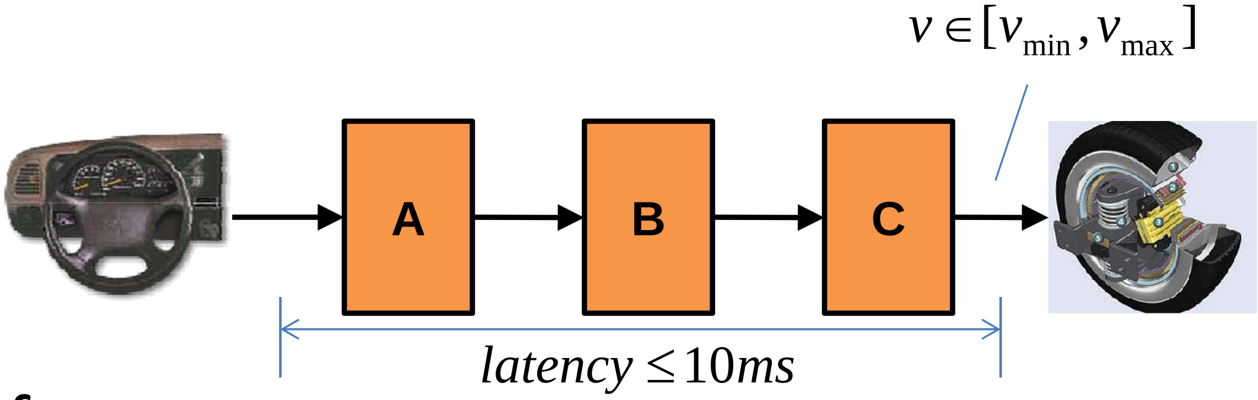
$v \in [v_{\min}, v_{\max}]$

$latency \leq 10ms$

- Actor interfaces
  - Performance properties (throughput, latency, …)

Which interface theories for CPS?

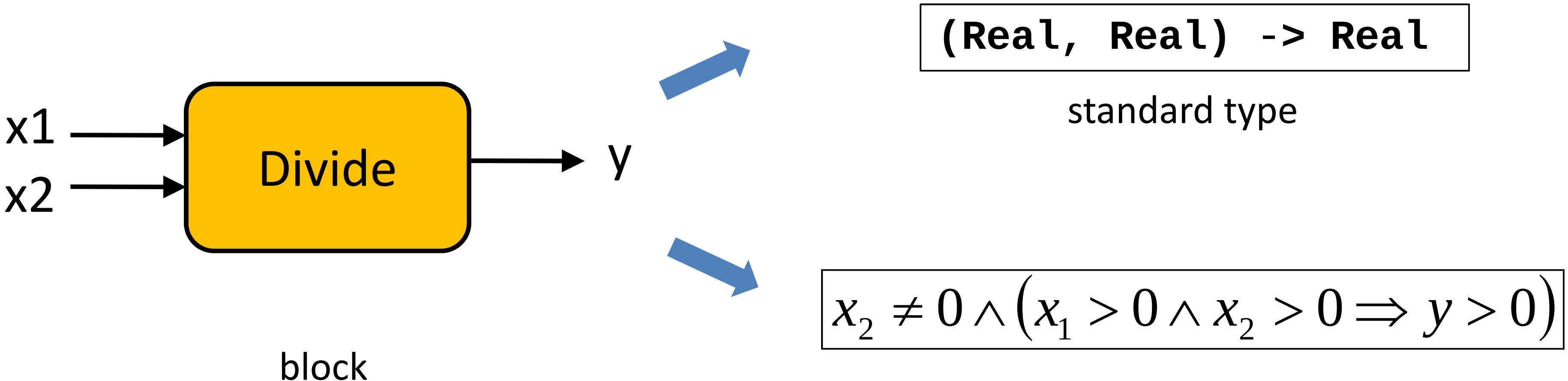
- Synchronous relational interfaces
  - Functional properties (correctness)



- Actor interfaces
  - Performance properties (throughput, latency, …)

[ACM TOPLAS 2011]

Interfaces for correctness



block

`(Real, Real) -> Real`

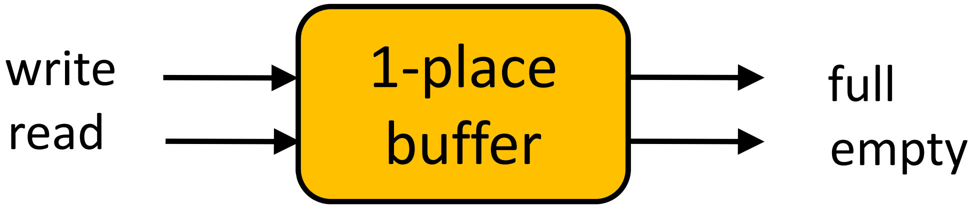
standard type

$$\boxed{x_2 \neq 0 \wedge \left(x_1 > 0 \wedge x_2 > 0 \Rightarrow y > 0\right)}$$

relational interface

- Akin to “behavioral types” [Liskov et al]

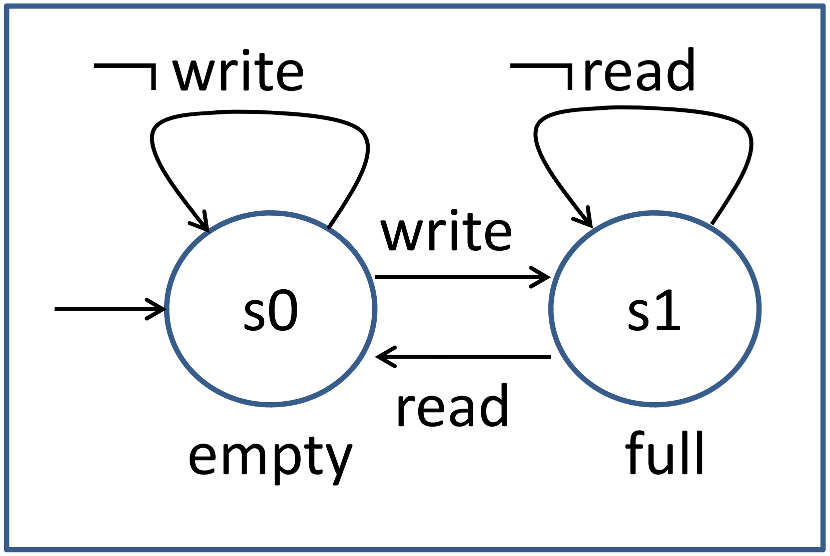
Interfaces for correctness: stateful (dynamic) interface


write
read
1-place buffer
full
empty

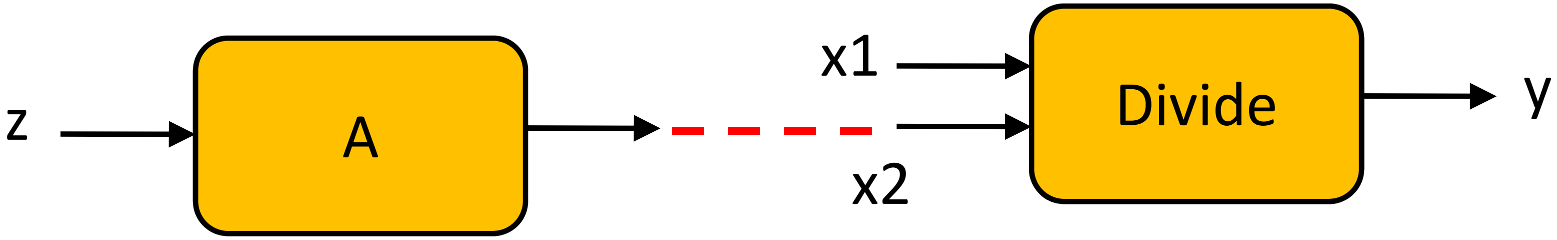
Static interface:
(holds at every round)

$$\neg(\text{empty} \wedge \text{full}) \\ \wedge \\ \neg(\text{write} \wedge \text{read}) \\ \wedge \\ \text{empty} \Rightarrow \neg\text{read} \\ \wedge \\ \text{full} \Rightarrow \neg\text{write}$$

Dynamic (state-dependent) interface:



Interfaces for correctness: compatibility checking = type checking

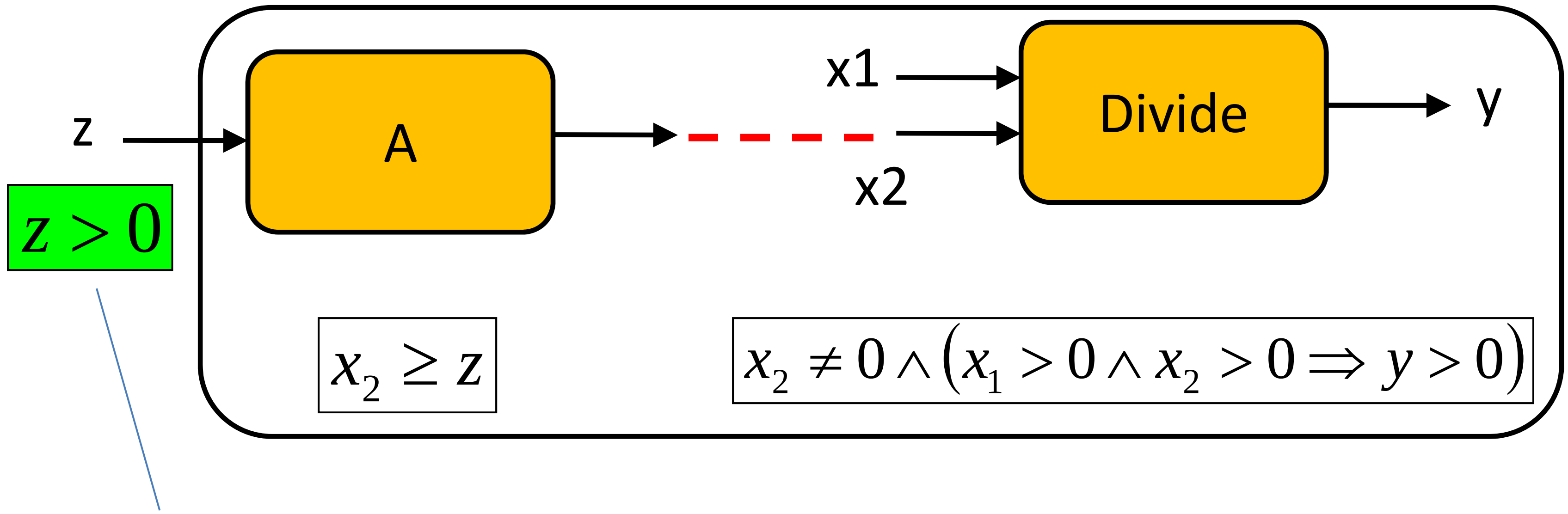


$$\boxed{x_2 \geq 0} \quad \not\Rightarrow \quad \boxed{x_2 \neq 0 \wedge \left(x_1 > 0 \wedge x_2 > 0 \Rightarrow y > 0\right)}$$

`Type error!`

Can be checked using SAT/SMT solvers

Interfaces for correctness: interface synthesis = type inference



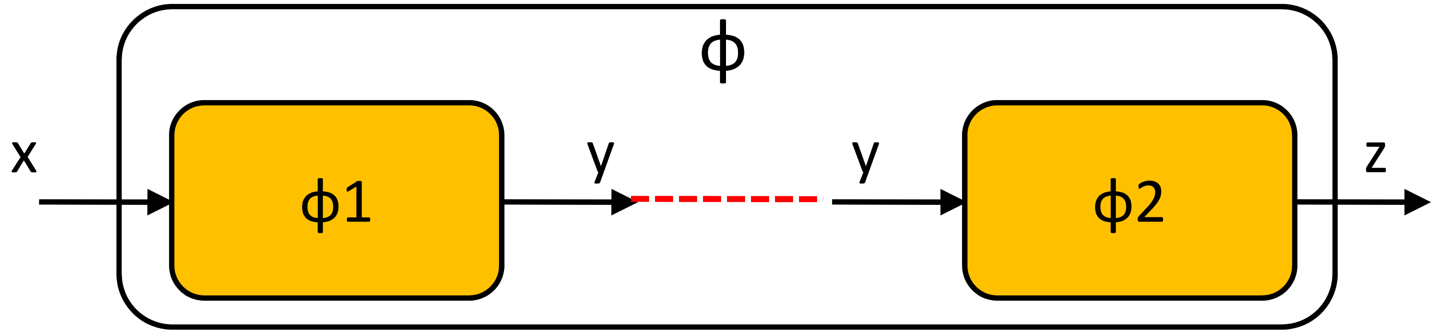
$z > 0$

$x_2 \geq z$

$x_2 \neq 0 \wedge \left(x_1 > 0 \wedge x_2 > 0 \Rightarrow y > 0\right)$

Automatically synthesized constraint

Interface synthesis:
“demonic” vs. standard composition



Standard: $\phi := \phi_1 \wedge \phi_2$

“Demonic”:

$$\phi := \phi_1 \wedge \phi_2 \wedge \overbrace{(\forall y : \phi_1 \Rightarrow in(\phi_2))}^{wp(\phi_1, in(\phi_2))}$$

$$in(\phi_2) := \exists z : \phi_2$$

Interfaces for correctness: refinement

$$\phi' \quad \leq \quad \phi$$

$$\overset{\text{def}}{=}$$

$$\boxed{\begin{array}{l} in(\phi) \Rightarrow in(\phi') \\ (in(\phi) \wedge \phi') \Rightarrow \phi \end{array}}$$

Interfaces for correctness: refinement

$$\boxed{\begin{array}{l} in(\phi) \Rightarrow in(\phi') \\ (in(\phi) \wedge \phi') \Rightarrow \phi \end{array}}$$

- Weaker than Liskov-Wing behavioral subtyping:

$$\boxed{\begin{array}{l} in(\phi) \Rightarrow in(\phi') \\ \phi' \Rightarrow \phi \end{array}}$$

Main results

- Preservation of refinement by composition (parallel, serial, restricted feedback):

If A’ ≤ A and B’ ≤ B, then A’ • B’ ≤ A • B.

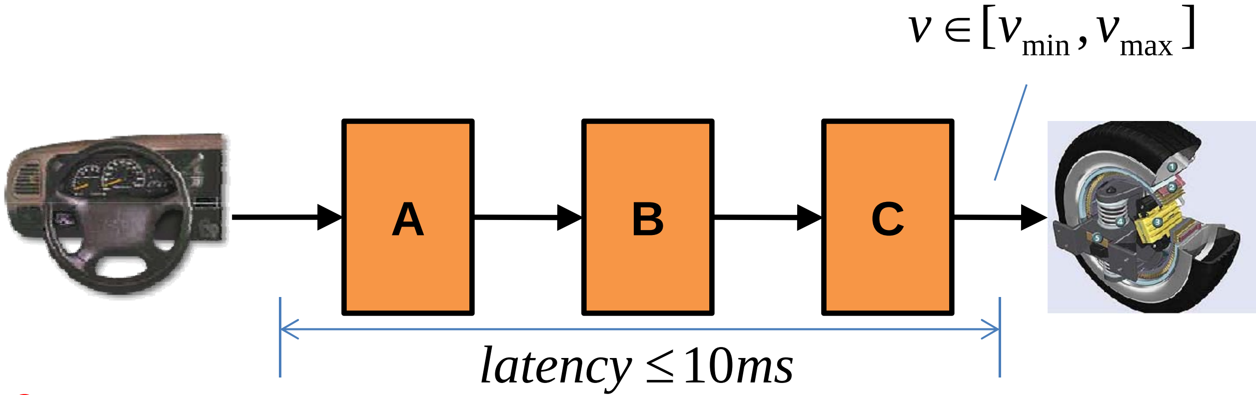
- Refinement <=> substitutability:

A’ can replace A in any context iff A’ ≤ A.

“full abstraction”:
refinement not stronger than necessary

Which interface theories for CPS?

- – Synchronous relational interfaces
    - Functional properties (correctness)

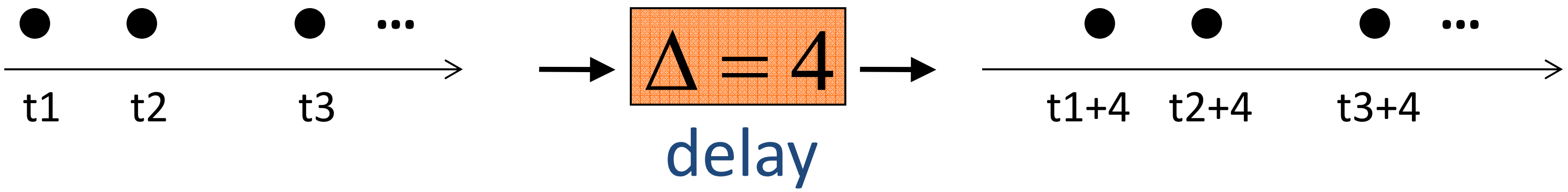


- – Actor interfaces
    - Performance properties (throughput, latency, …)

[HSCC 2011]

Interfaces for performance

- Relations between I/O **timed event streams**
  - No values associated with events


... t1 t2 t3

$\Delta = 4$

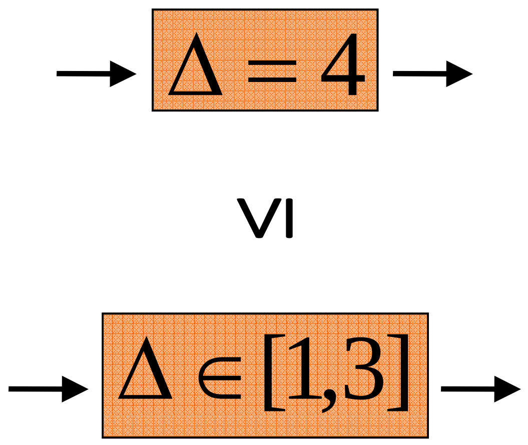
delay

... t1+4 t2+4 t3+4

Interfaces for performance refinement: “the earlier the better”

deterministic delay:

nondeterministic delay:



$$\rightarrow \boxed{\Delta = 4} \rightarrow$$

$$\rotatebox{90}{$\leq$}$$

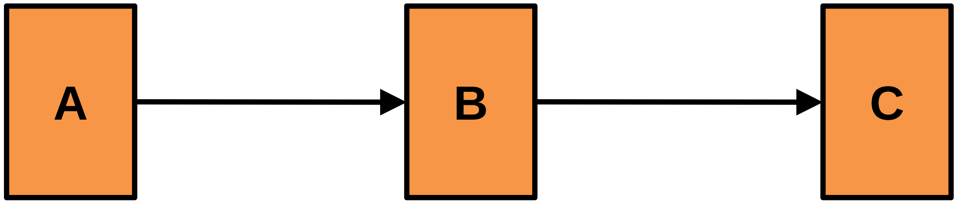
$$\rightarrow \boxed{\Delta \in [1,3]} \rightarrow$$

- vs. standard refinement = inclusion of behaviors = implementations more deterministic than specs

What it buys us

easier
to analyze,
verify, …

time-deterministic model



preserves
worst-case
performance
(throughput,
latency)

VI

time-nondeterministic system

Conclusions

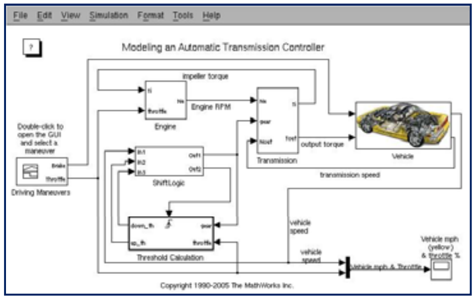
- **Interfaces** = abstractions of components
  - Keep necessary information, hide the rest

- No unique, one-size-fits-all interface model / theory

- General principles:
  - Bottom-up interface synthesis for hierarchical languages
  - **Refinement** for incremental design

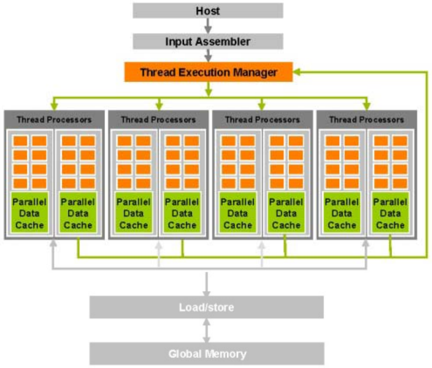
Thank you

- Questions?

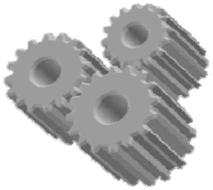


Richer languages:
concurrency, time,
robustness, reliability,
energy, security, …

System Compiler



more complex
execution platforms:
networked, distributed,
multicore, …



more powerful analyses:
model-checking, WCET analysis, schedulability,
performance analysis, reliability analysis, …