Modular code generation from synchronous models:

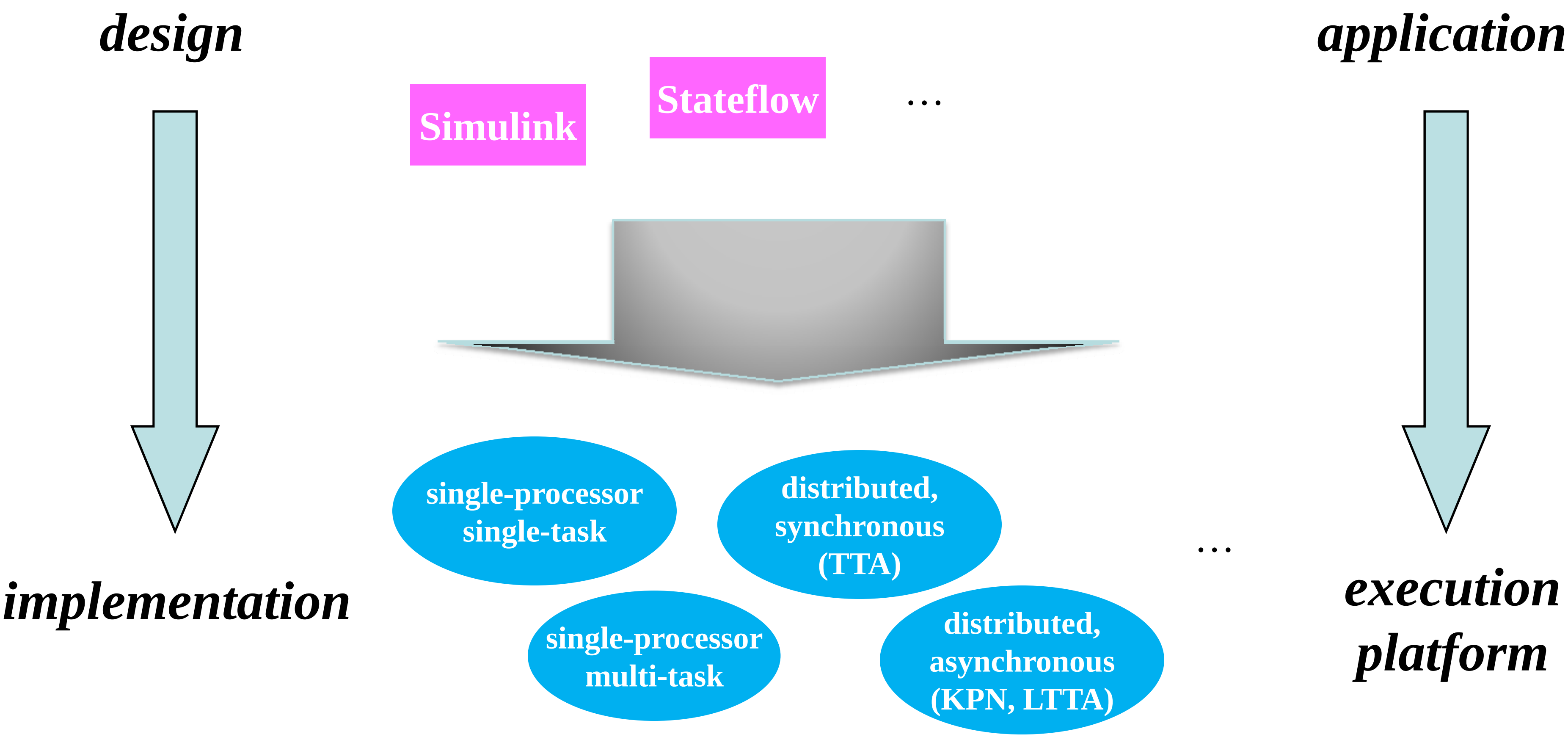
modularity vs. **reusability** vs. **code size**

Stavros Tripakis

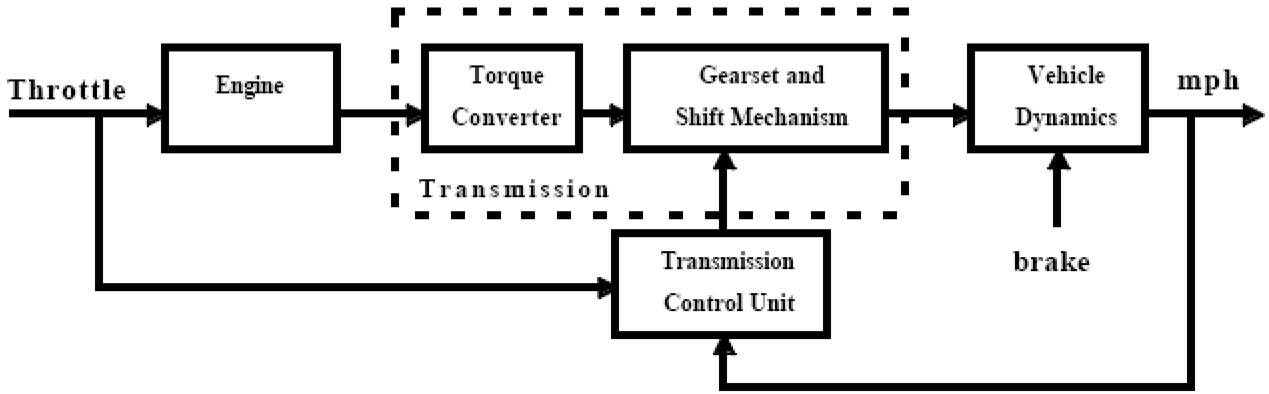
Joint work with

Roberto Lublinerman, Penn State

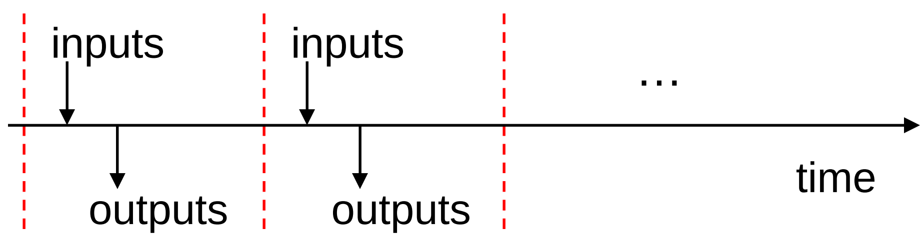
Semantics-preserving implementation of "high-level" models



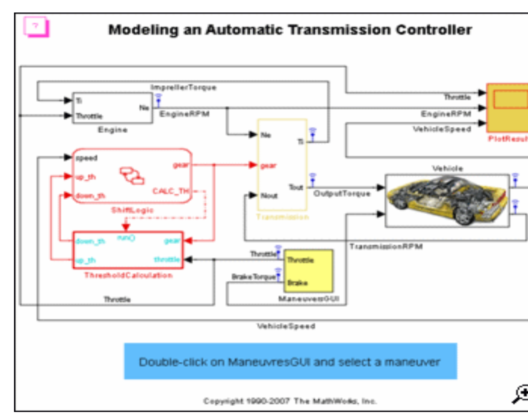
Synchronous block diagrams



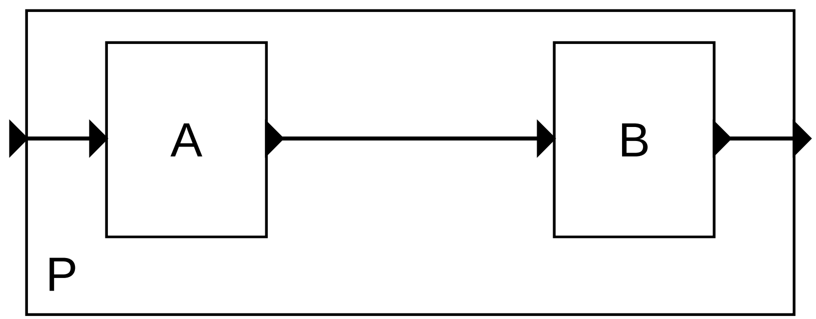
- Fundamental model behind (discrete-time) **Simulink**, SCADE, synchronous languages (Lustre, Esterel, …)
- Widely used in **embedded systems**
- **Synchronous**, deterministic semantics:



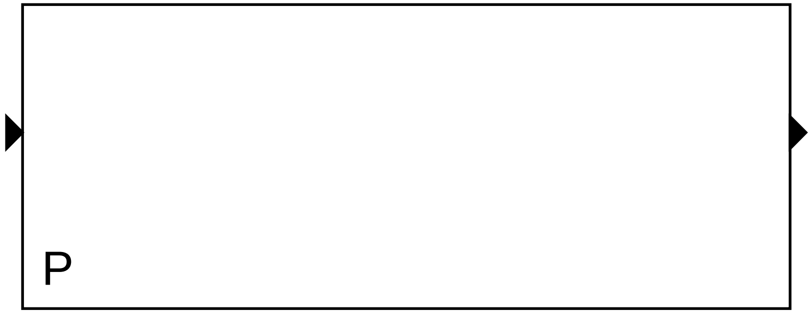
Copyright The Mathworks



Hierarchy



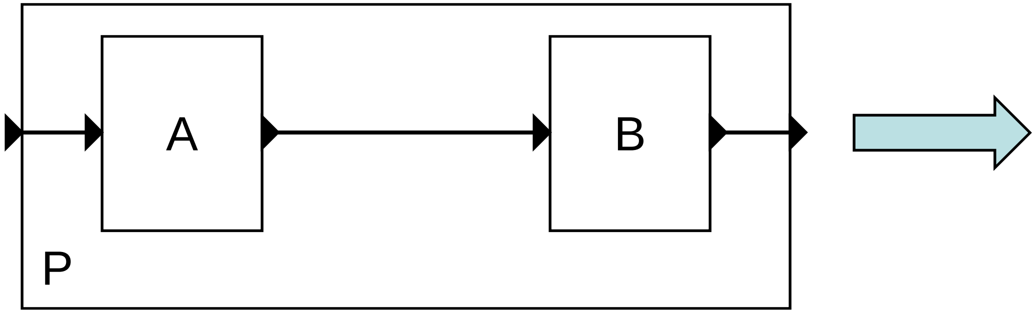
Hierarchy



Fundamental modularity concept

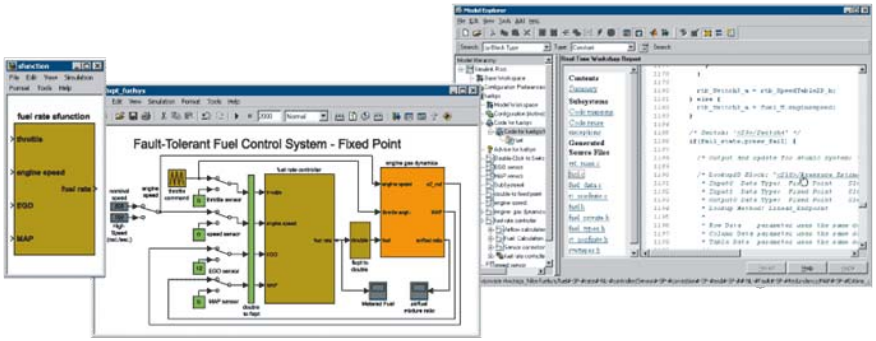
Code generation

- Generate imperative code (in C, C++, Java, …) that implements the semantics

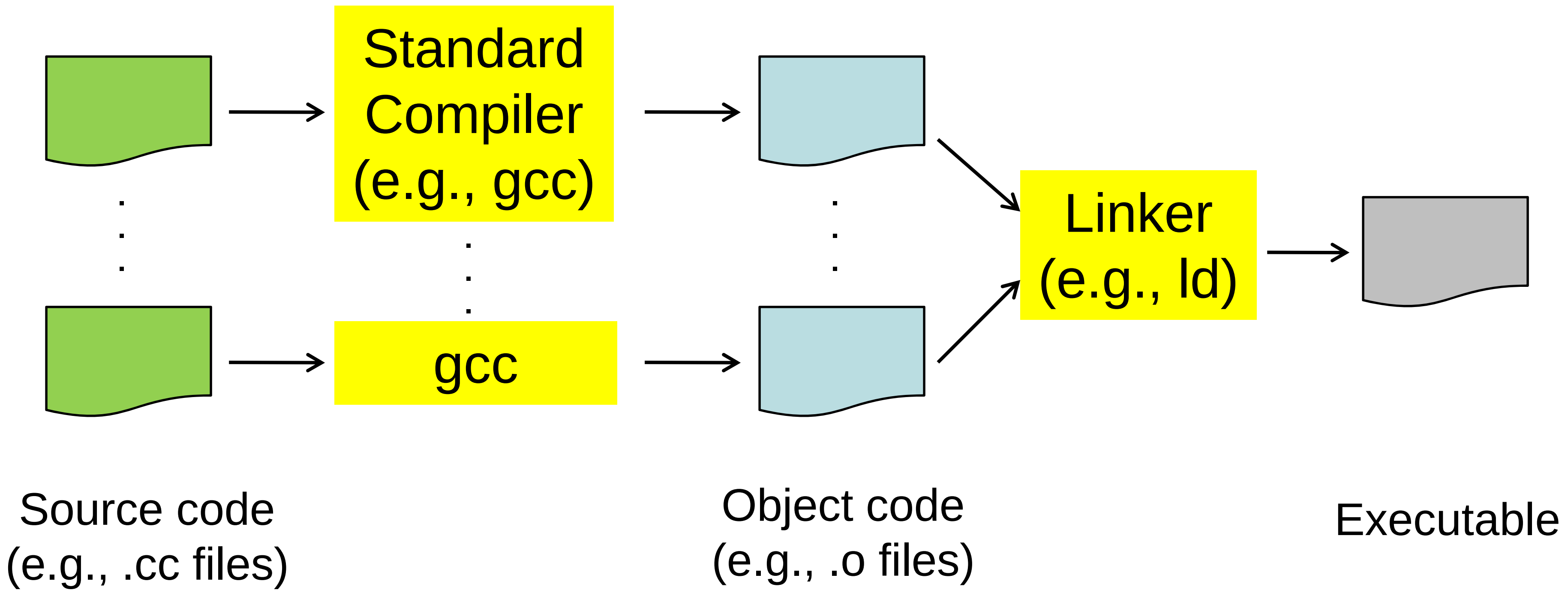



```
P.step( in ) returns out
{
  tmp := A.step( in );
  out := B.step( tmp );
  return out;
}
```

- Code may be used for simulation, embedded control (“X-by-wire”), …
  - SCADE
  - Real-Time Workshop
  - …



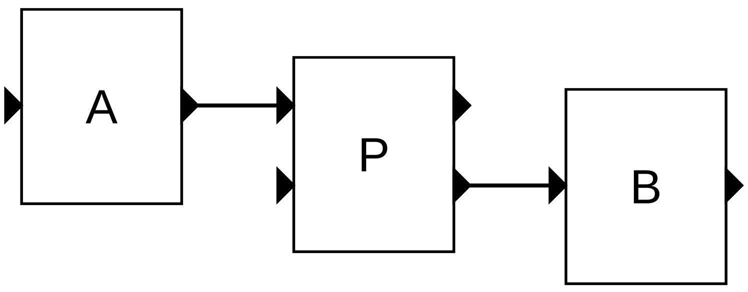
Separate compilation



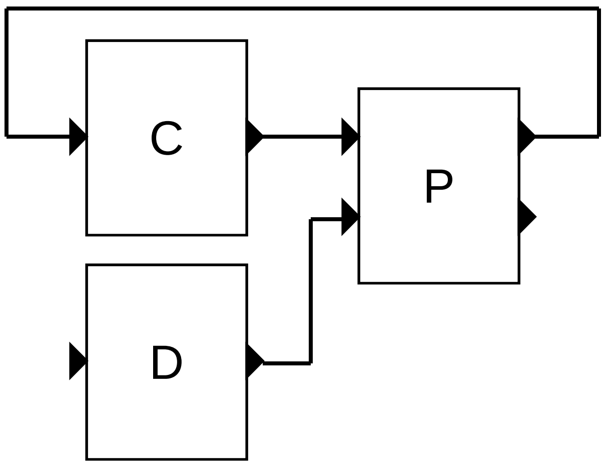
We want to do the same for synchronous block diagrams

Modular code generation

- Goal: generate code for a given block P
- Code should be independent from context:



Will P be connected like this?



…or like that?

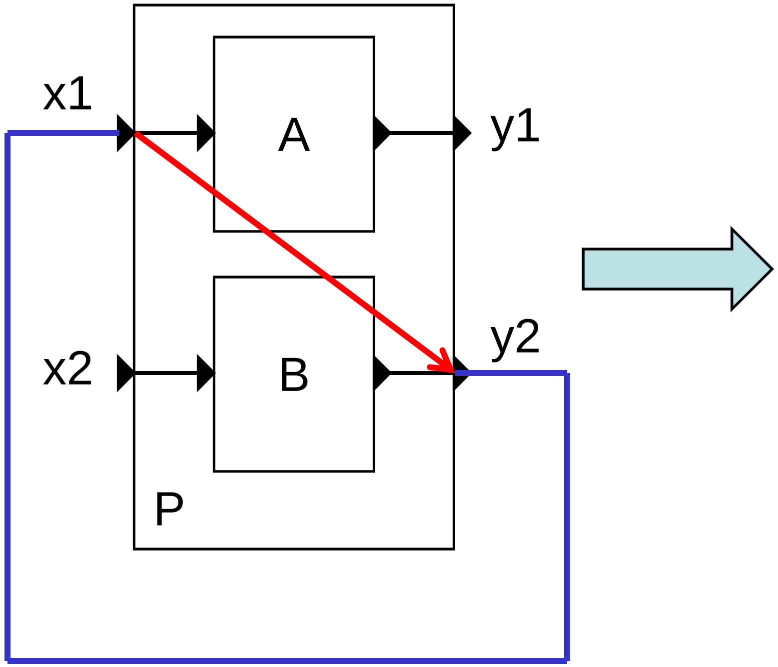
  - Enables component-based design (c.f., AUTOSAR)

Problem with current approaches: “monolithic” code

False I/O dependencies

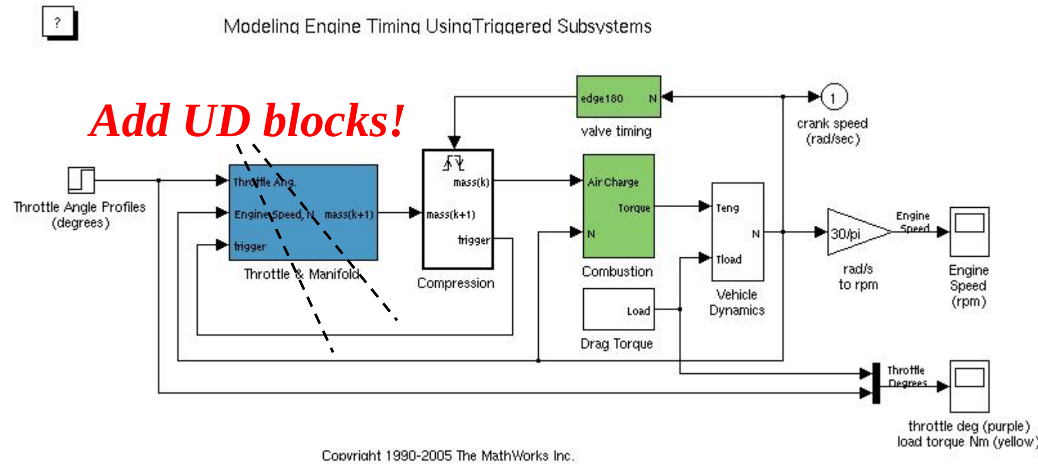
=>

code not usable in some contexts



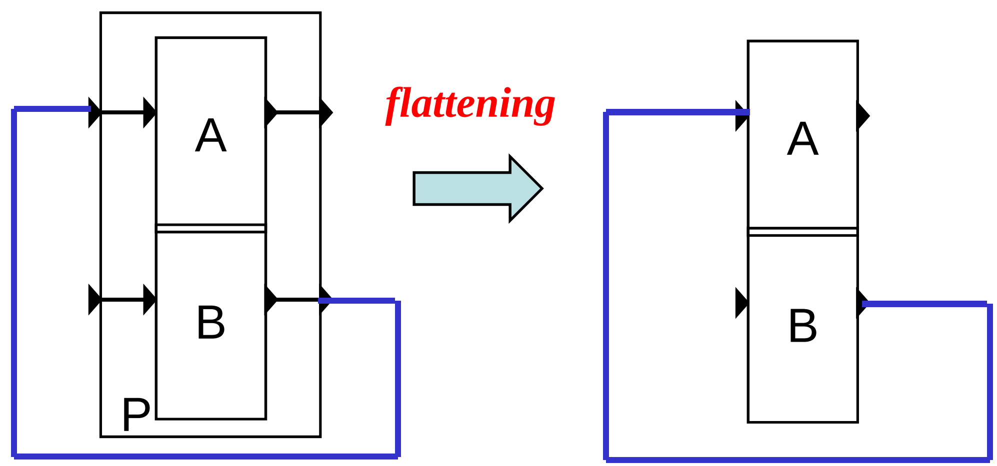

```
P.step(x1, x2) returns (y1, y2)
{
      y1 := A.step( x1 );
      y2 := B.step( x2 );

      return (y1, y2);
}
```

Code generation – state of the art



- Either restrict diagram:
  - Break cycles at each level with ***unit-delays*** (SCADE)

- Or flatten (Simulink/RTW)
  - Remove diagram hierarchy



- Problem sometimes claimed impossible to solve [Girault'05]

Other approaches

- Dynamic fix-point computation [Edwards-Lee'03]:
  - Start with "bottom" (undefined value) assigned to all wires in the diagram
  - Keep calling "step()" functions until you find a fix-point
    - There is a unique fix-point but it may contain "bottom" values
  - One would like to check that the fix-point does not contain "bottom" values

- Can do this by checking whether diagram is constructive [Malik'94, Berry et al.'96]
  - Undecidable in general, expensive otherwise
  - Needs semantic information:
    - What is the function that this block computes?
    - Contrary to our black-box component view

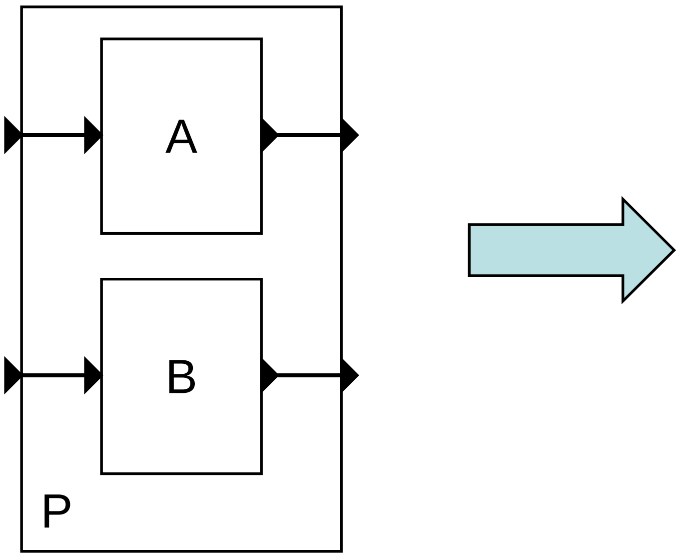
[DATE’08, RTAS’08, POPL’09]

Our solution

- Modular:
  - No more flattening
- General:
  - No restrictions: handles all diagrams that can be handled by flattening
- Not one, but many solutions:
  - Explore different trade-offs

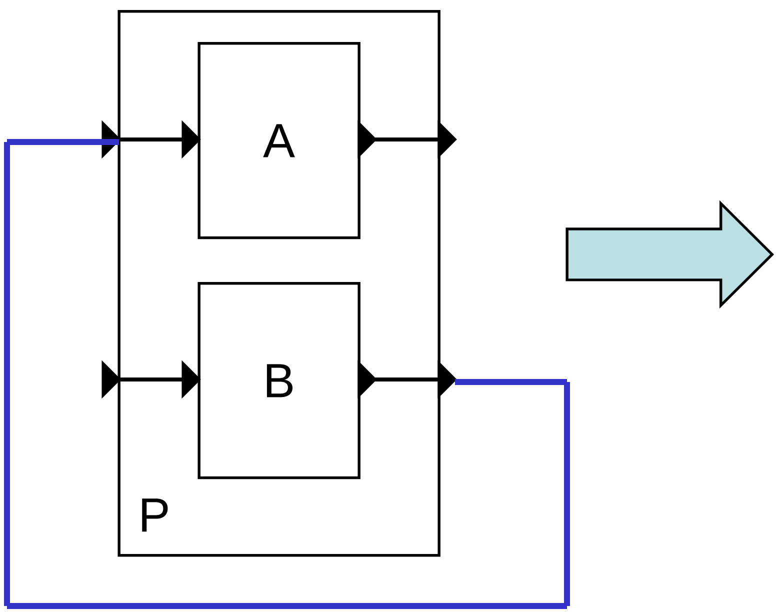
How do we do it?

- Generate for each block a PROFILE = INTERFACE
- Interface may contain MANY functions




```
P.step1( in1 ) returns out1 {
   return A.step( in1 );
}


P.step2( in2 ) returns out2 {
   return B.step( in2 );
}
```

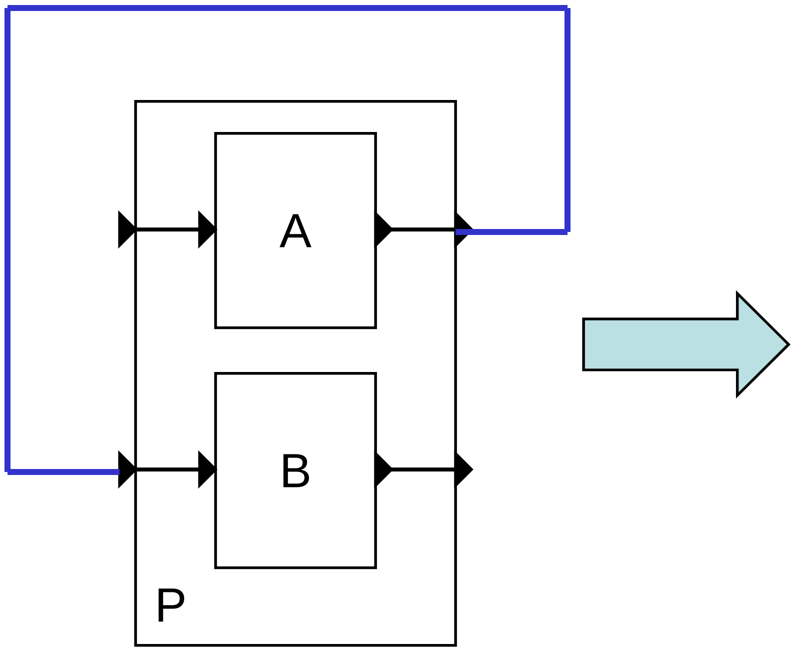
How do we do it?




```
P.step1( in1 ) returns out1 {
   return A.step( in1 );
}

P.step2( in2 ) returns out2 {
   return B.step( in2 );
}
```

How do we do it?

The function call order depends on the usage of the block




```
P.step1( in1 ) returns out1 {
   return A.step( in1 );
}

P.step2( in2 ) returns out2 {
   return B.step( in2 );
}
```

Number of interface functions

Modularity vs. Reusability

Set of allowed contexts

Modularity crucial for:
(1) Scalability
(2) IP issues

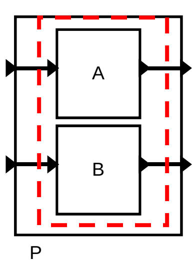
more modular, less reusable

more reusable, less modular

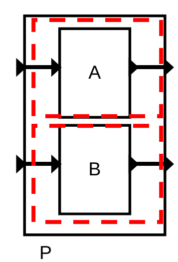
Modularity becomes quantifiable!

fewer interface functions

more interface functions



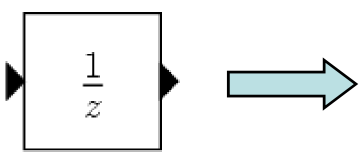

```
class P {
  public Pstep( in1, in2  ) returns out1, out2;

  Pstep( in1, in2 ) {
    return (Astep( in1 ), Bstep( in2 ) );
  }

}
```




```
class P {
  public Pstep1( in1 ) returns out1;
  public Pstep2( in2 ) returns out2;

  Pstep1( in1 ) {
    return Astep( in1 );
  }

  Pstep2( in2 ) {
    return Bstep( in2 );
  }
}
```

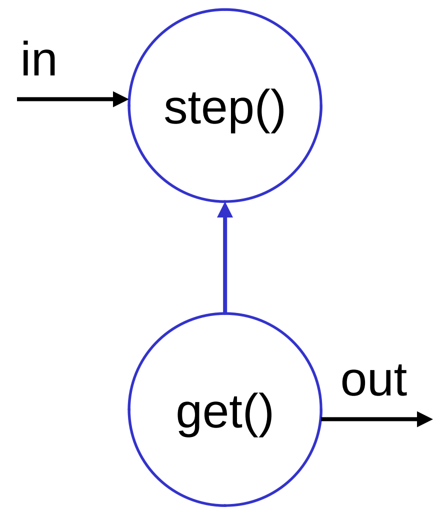
Profile dependency graphs

- Profile = Interface functions + DEPENDENCY GRAPH
- Graph encodes interface usage constraints



unit-delay

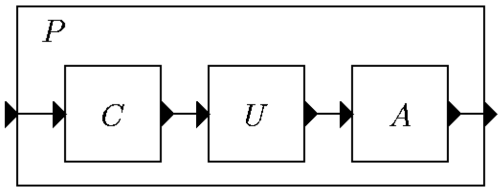
```
class UnitDelay {

  private state;

  step( in ) returns void {
     state := in;
  }

  get() returns out {
     return state;
  }
}
```



PROFILE DEPENDENCY GRAPH

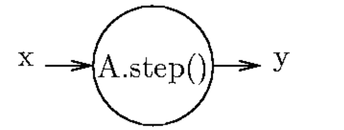
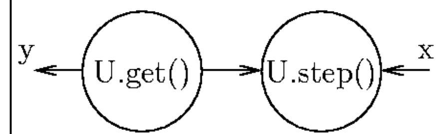
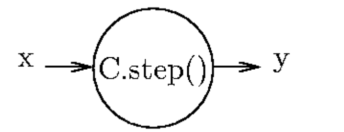
Overall method

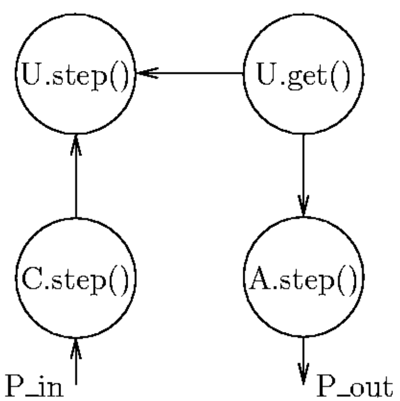
Input 1



A macro block P
and its internal diagram

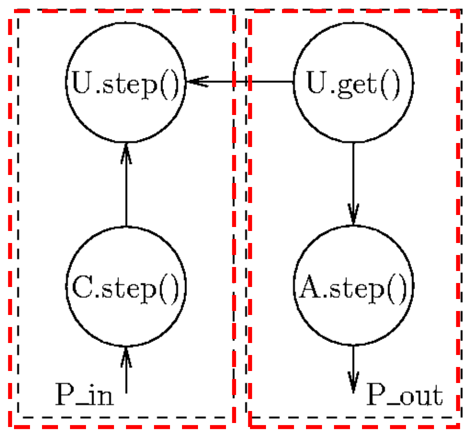
Input 2

	Interface functions	Profile dependency graphs
Profile of A: (combinational)	A.step(x) returns y;	
Profile of U: (Moore-sequential)	U.get() returns y; U.step(x) returns void;	
Profile of C: (combinational)	C.step(x) returns y;	



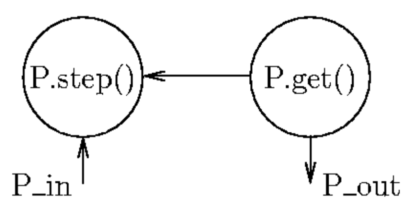
SDG of P

scheduling dependency graph



SDG of P clustered in two sub-graphs

Clustering



Resulting interface functions
and PDG of P

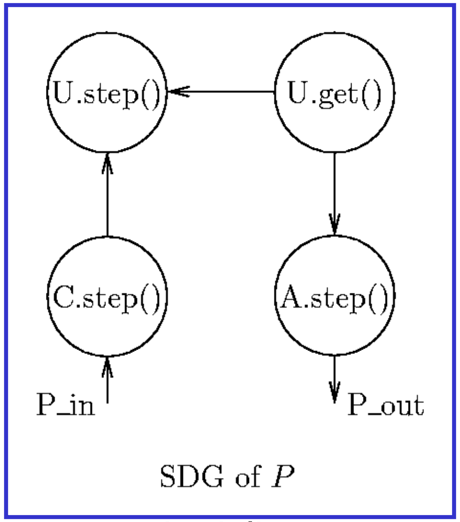
profile dependency graph

Trade-offs

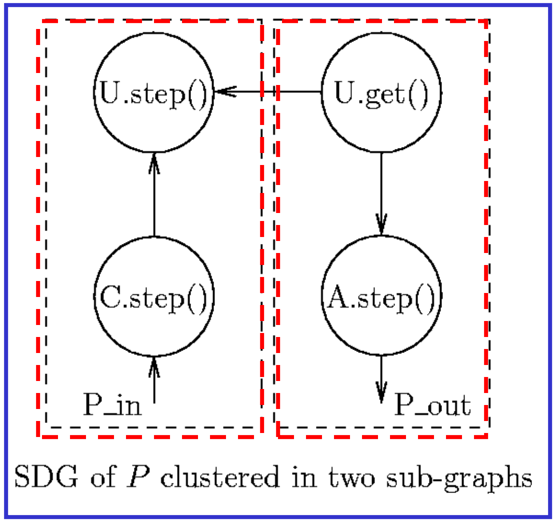
different clusterings => different trade-offs

Currently have 3 clustering algorithms:

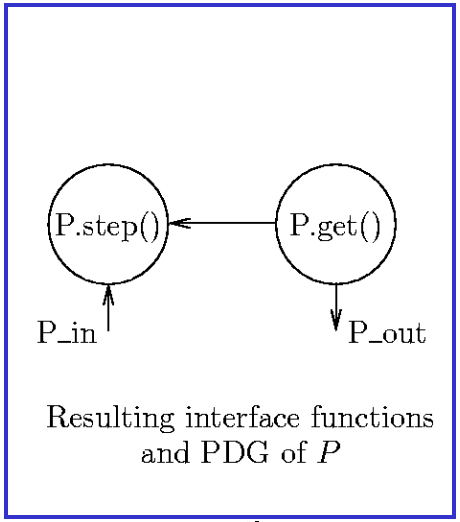
- Step-get clustering: 1 or 2 methods per block (classic)
- Dynamic clustering
- Optimal disjoint clustering



scheduling dependency graph



Clustering



profile dependency graph

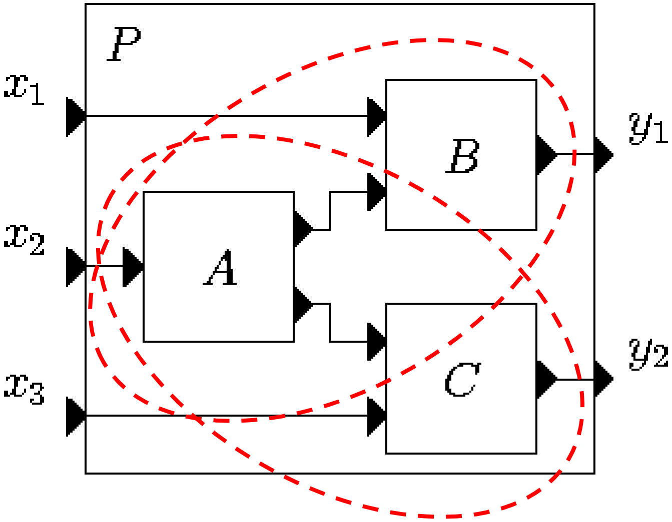
[DATE'08]

Dynamic clustering

- Group outputs w.r.t. input dependencies
- For each group, compute transitive fan-in

- Achieves:
  - Maximal reusability: code can be used in ANY context
  - Optimal modularity: minimal number of interface functions
  - Bound: <= N+1 functions
    - N: number of block outputs

Optimal modularity => overlapping clusters



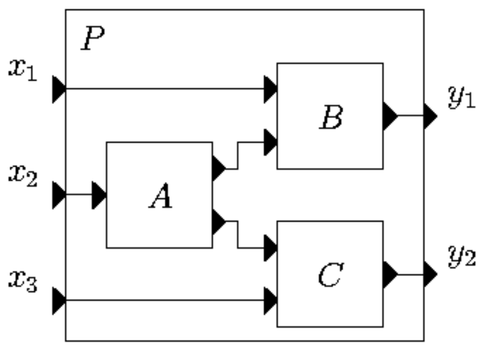
A macro block P

2 interface functions (optimal)

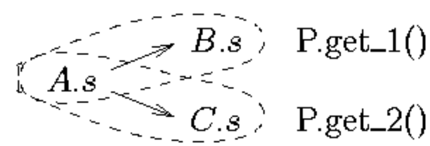
Overlapping clusters => extra code for “dynamic” scheduling

```
P.get1( x1, x2 ) returns y1 {
  if (cA = 0) {
    (z1, z2) := A.step( x2 );
  }
  cA := (cA + 1) modulo 2;
  y1 := B.step( x1, z1 );
  return y1;
}
```

```
P.get2( x2, x3 ) returns y2 {
  if (cA = 0) {
    (z1, z2) := A.step( x2 );
  }
  cA := (cA + 1) modulo 2;
  y2 := C.step( z2, x3 );
  return y2;
}
```



A macro block P



Clustered SDG of P and corresponding interface functions

Overlapping clusters
=> code replication

```
P.get1( x1, x2 ) returns y1 {
  if (cA = 0) {
    (z1, z2) := A.step( x2 );
  }
  cA := (cA + 1) modulo 2;
  y1 := B.step( x1, z1 );
  return y1;
}
```

```
P.get2( x2, x3 ) returns y2 {
  if (cA = 0) {
    (z1, z2) := A.step( x2 );
  }
  cA := (cA + 1) modulo 2;
  y2 := C.step( z2, x3 );
  return y2;
}
```

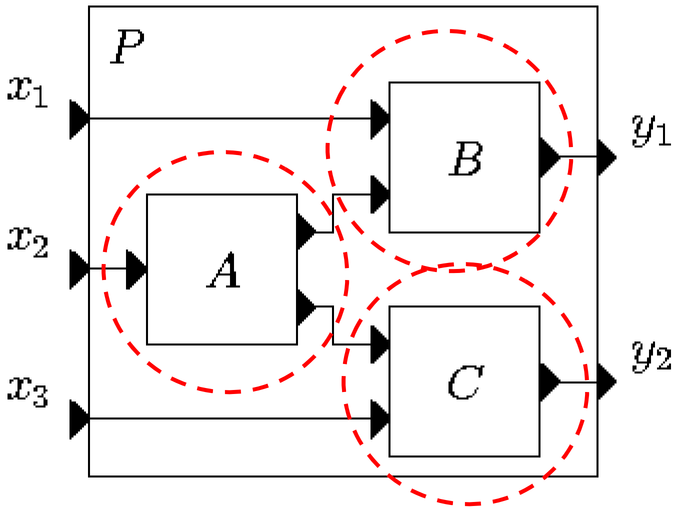
- Code size crucial for embedded systems
- We want to minimize it =>
- We want **disjoint clusters**

[POPL'09]

Optimal disjoint clustering

- **Optimal disjoint clustering** problem:
  - How to cluster/partition a DAG into a minimal number of disjoint clusters, without introducing new input-output dependencies
- Optimal disjoint clustering is NP-complete
  - *With Christian Szegedy (Cadence labs)*
- Good news:
  - Can be reduced to a SAT problem (for given # of clusters)
  - Very efficient in practice!

Another trade-off: modularity vs. code size



A macro block P

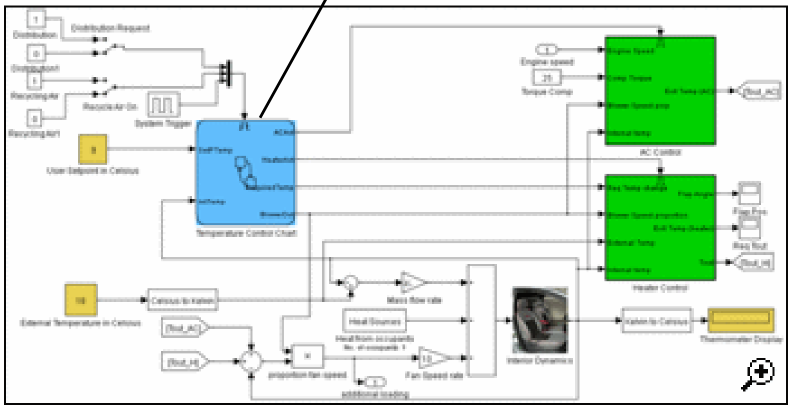
3 interface functions (sub-optimal)

Extension to triggered and timed diagrams

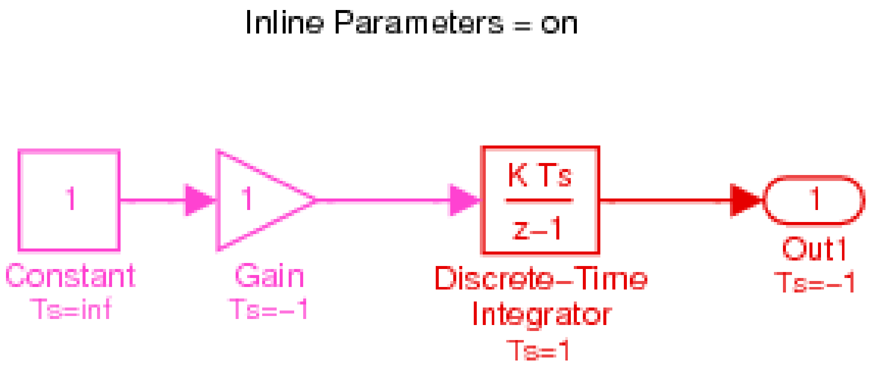
[RTAS'08]

- Triggers: found in Simulink, SCADE, synchronous languages, …
- Sample times = static, periodic triggers

Triggered block



Simulink/Stateflow diagram

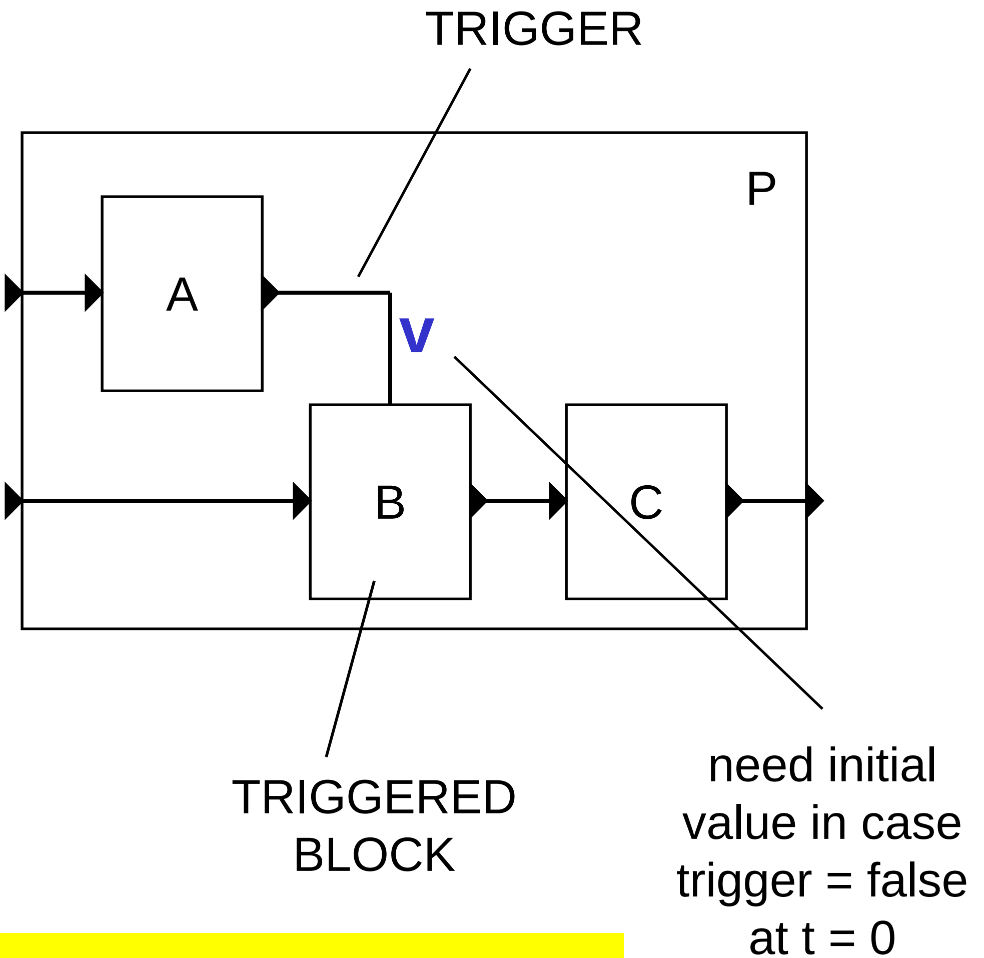


Sample time

Triggered diagrams

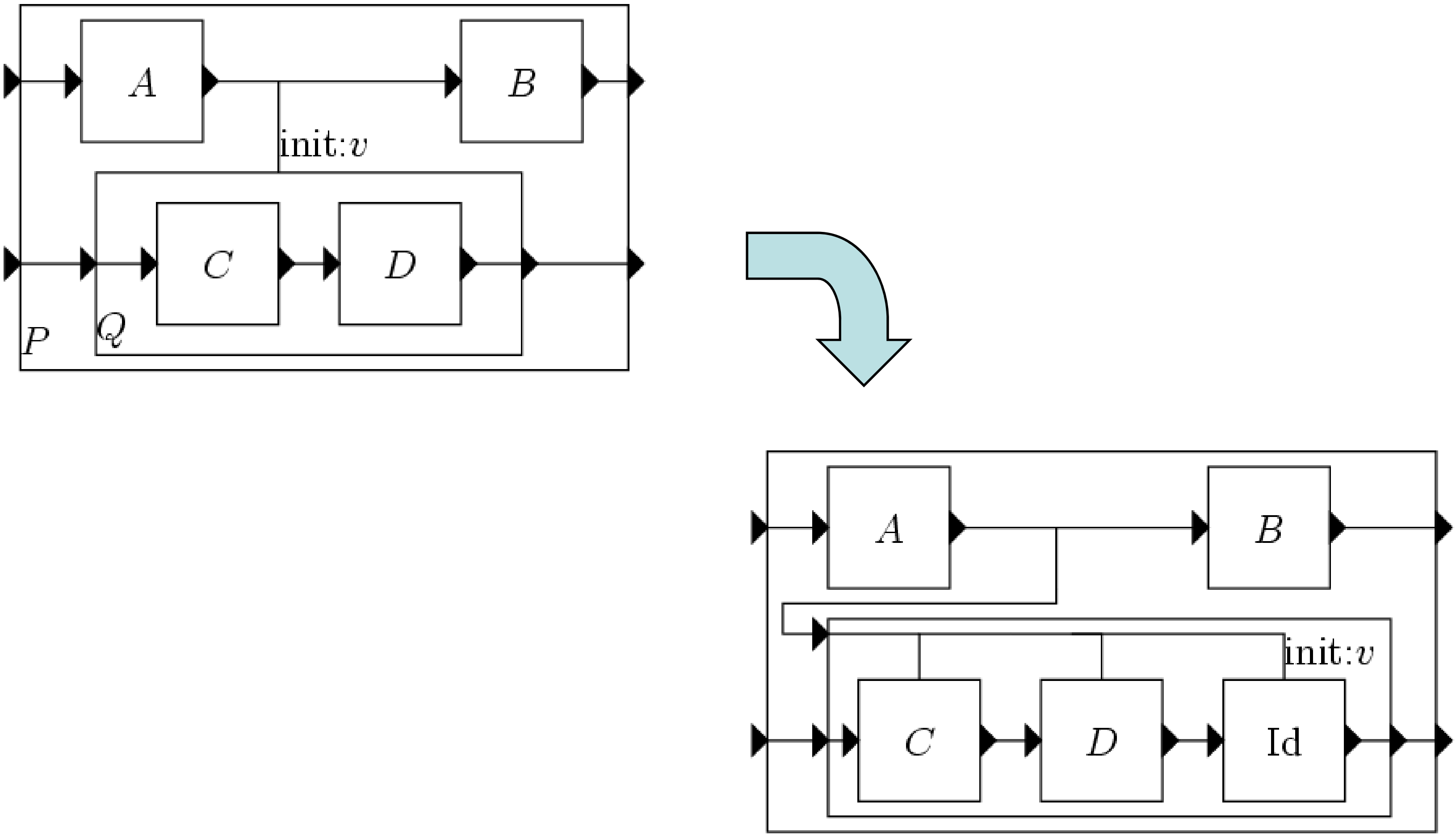
multi-rate models:

- B executed only when trigger = true
- All signals "present" always
- But not all updated at the same time
- E.g., output of B updated only when trigger is true

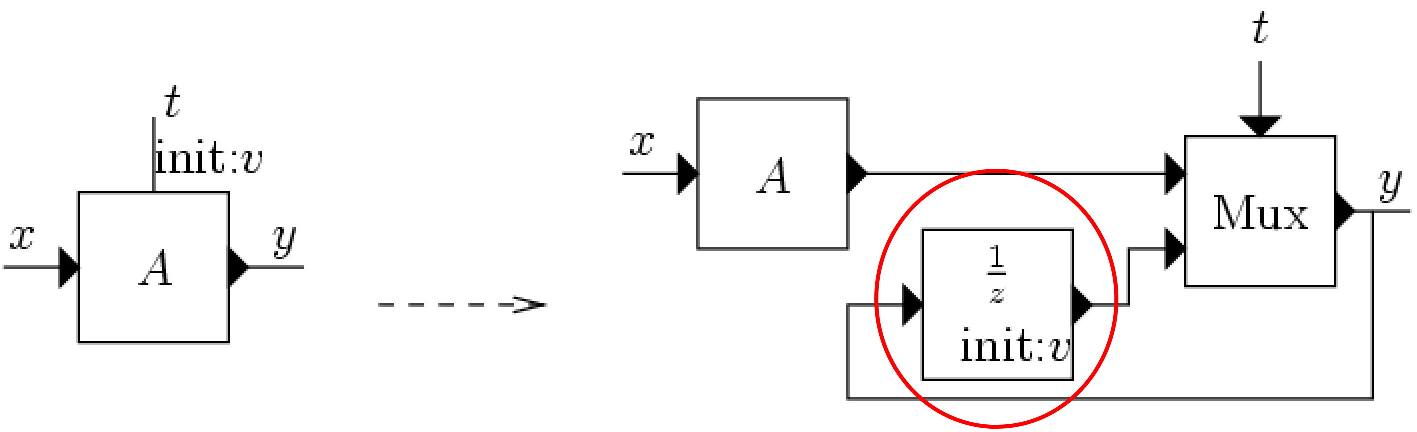


Question: do triggers increase expressiveness?

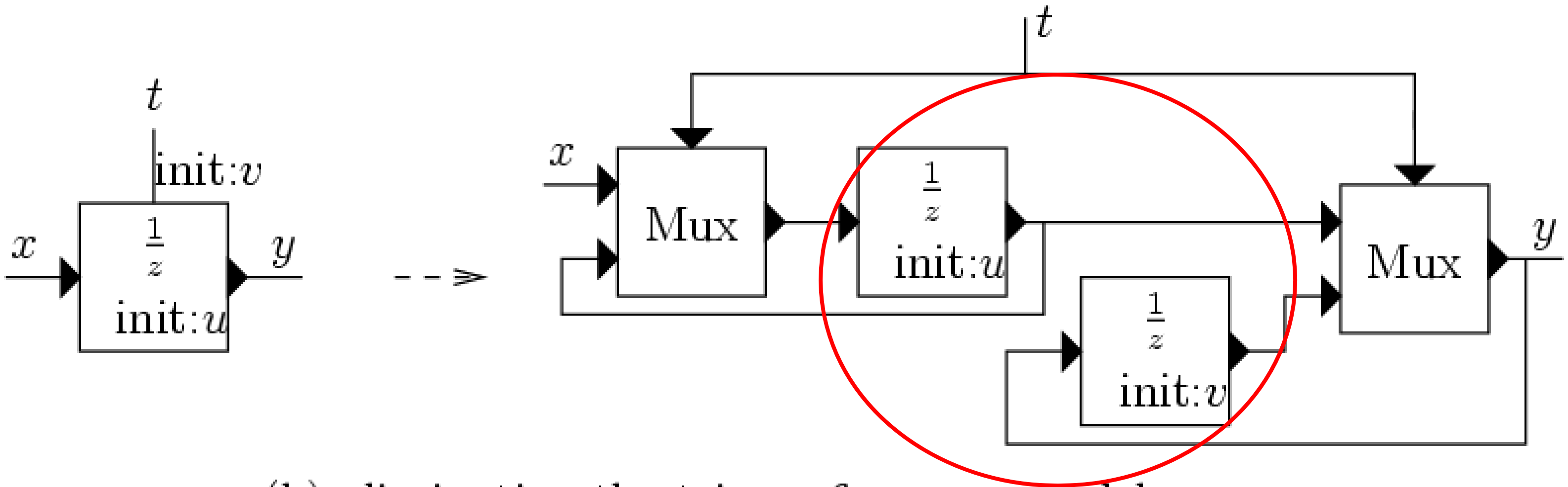
Trigger elimination



Trigger elimination: atomic blocks



(a) eliminating the trigger from a combinational atomic block

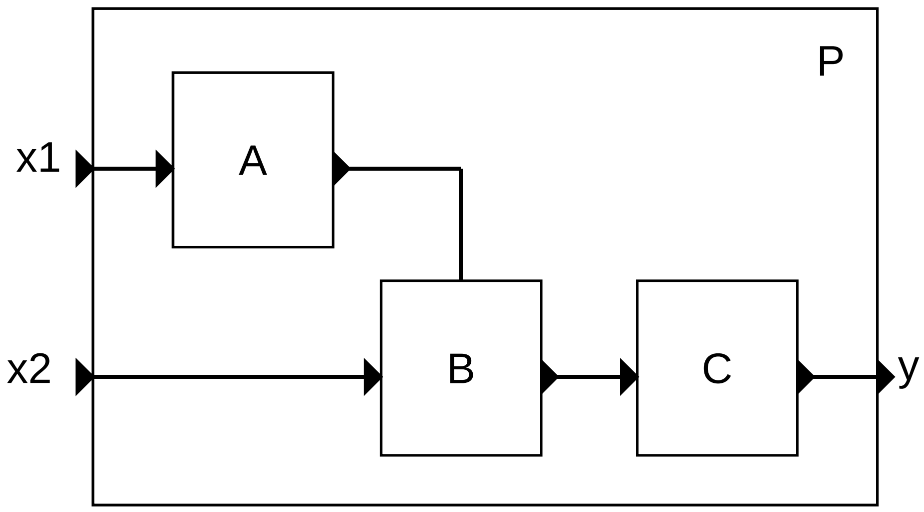


(b) eliminating the trigger from a unit-delay

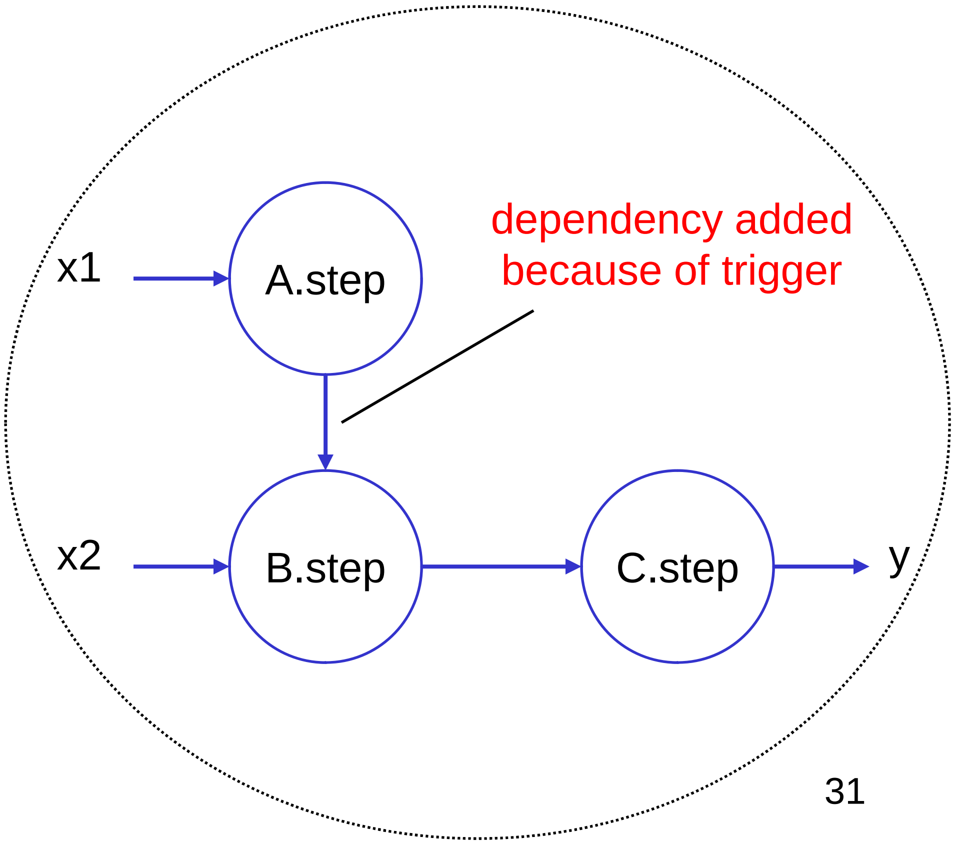
Trigger elimination: summary

- Can be done: preserves the semantics
- But:
  - It requires flattening => it destroys modularity
  - (must propagate triggers top-down => “open the boxes”)
- Solution:
  - Handle triggers directly, without eliminating them

Handling triggered diagrams directly

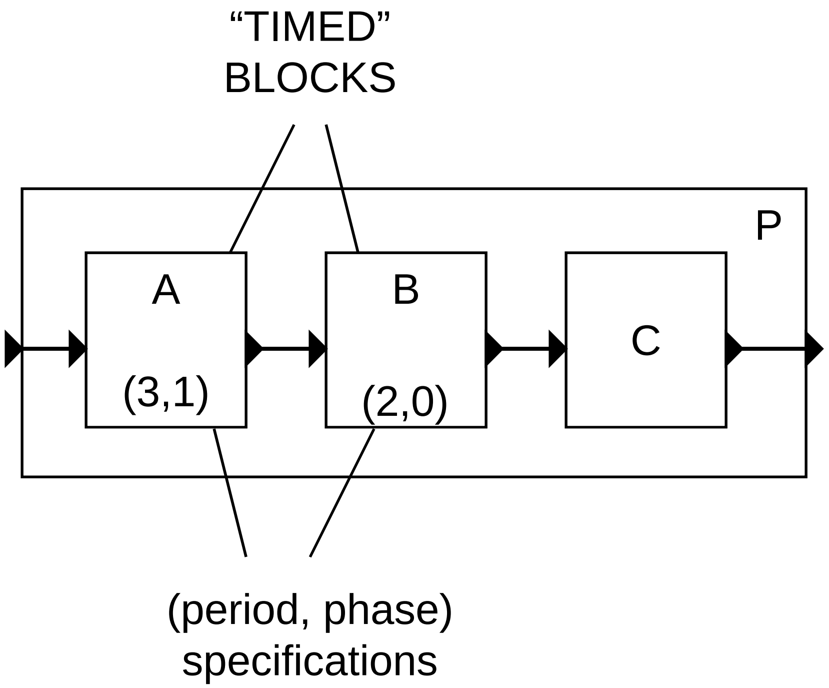


Scheduling Dependency Graph of P:

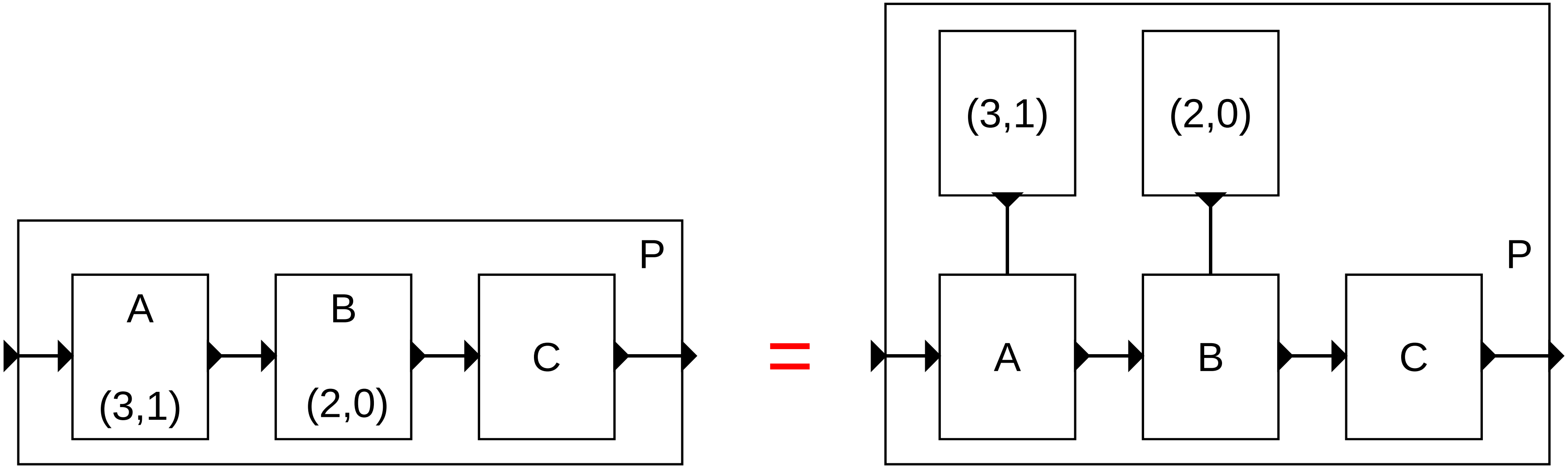


Timed diagrams

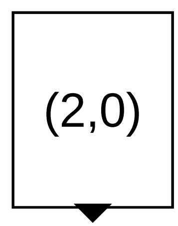
“static” multi-rate models



Timed diagrams = “static” triggered diagrams



where

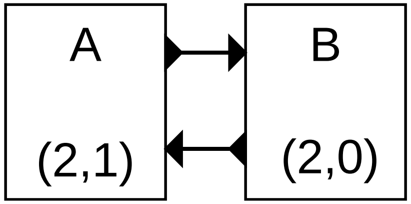


produces: true, false, true, false, …

Handling timed diagrams

- Could treat them as triggered diagrams
- But we can do better:
- Exploit the **static information** that timed diagrams provide:
  - To identify cases of false dependencies => accept more diagrams
  - To avoid firing blocks unnecessarily => more efficient code

Identifying false dependencies

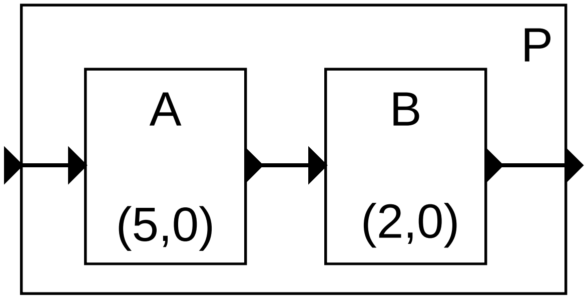


A and B are never active at the same time

=>

Both dependencies are false

Eliminating redundant firings



Q: how often should P be fired?

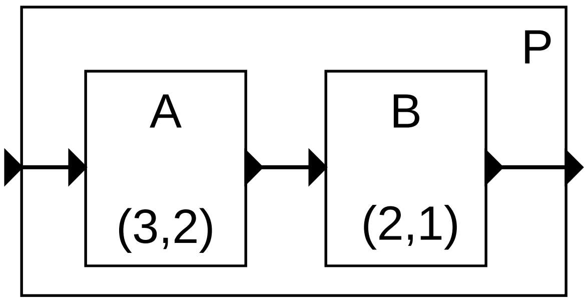
Simple answer: every GCD(5,2) = 1 time unit = at every “clock cycle”

Better answer: at cycles {0,2,4,5,6,8,10, …} = only when it needs to

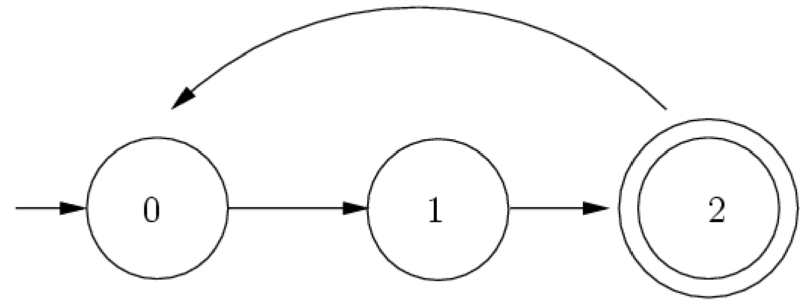
Problem: (period,phase) representation not closed under union

Solution: Firing Time Automata

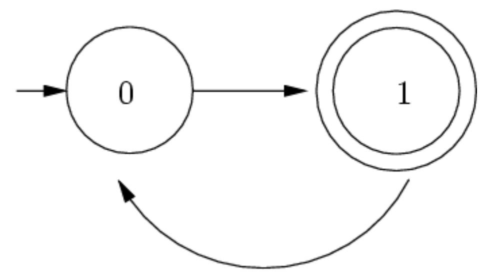
Firing Time Automata

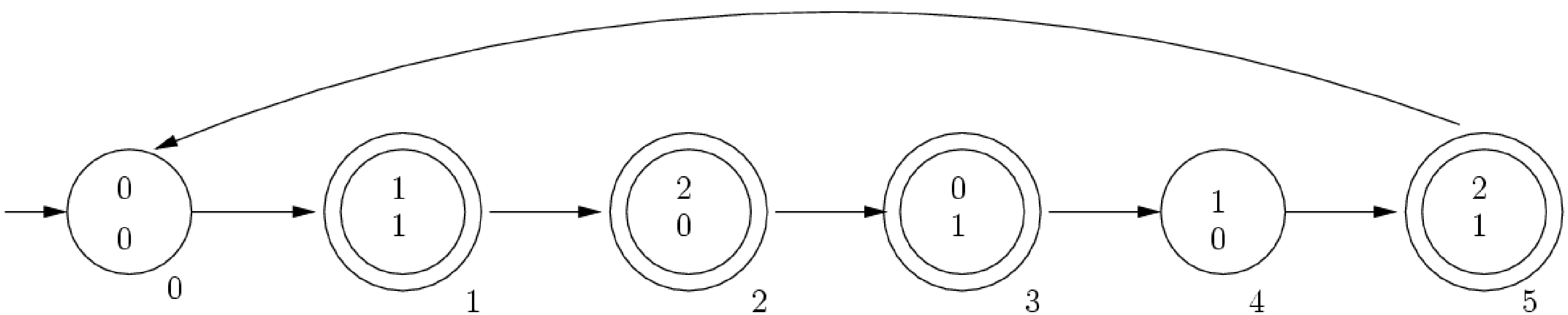


A



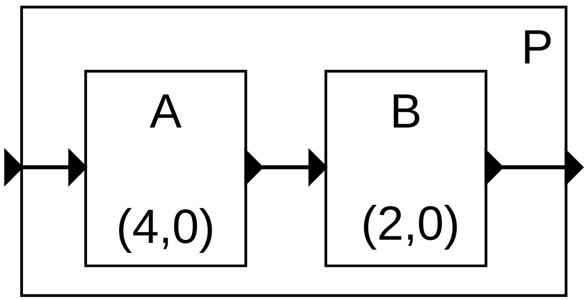
B

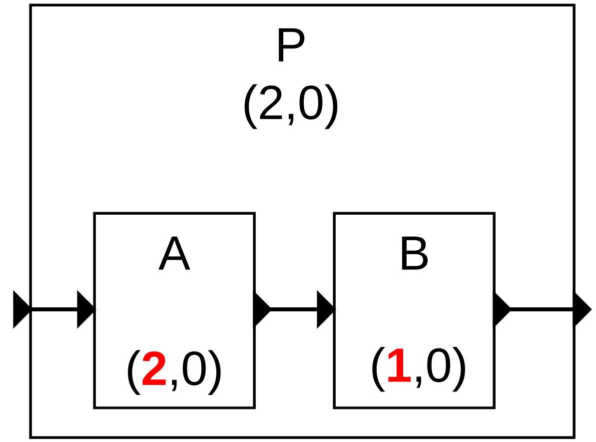




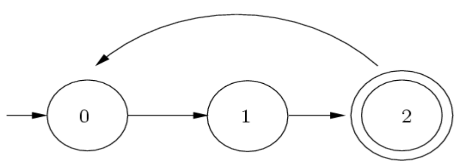
A ∪ *B*

FTA division

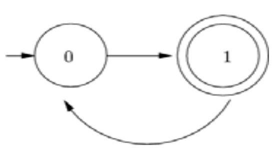


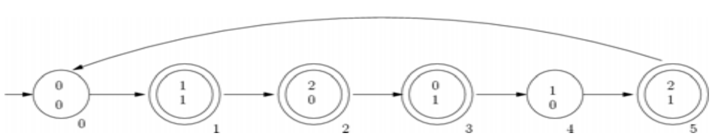


A



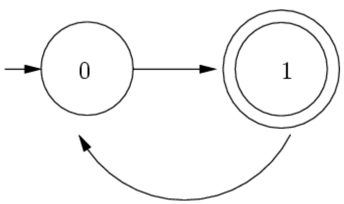
B



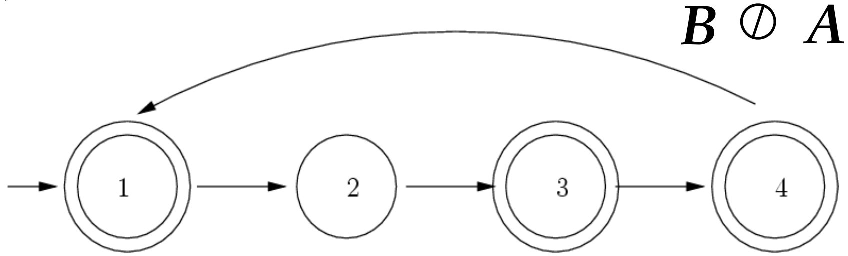


$A \cup B$

$A \oslash A \cup B$



$B \oslash A \cup B$



FTA union, division, multiplication

$$\begin{aligned} \boxed{A \cup B} &= \left(S_A \times S_B, \left(s_0^A, s_0^B\right), \left\{(s_A, s_B) \,|s_A \in F_A \vee s_B \in F_B\right\}, T_{A \cup B}\right) \\ T_{A \cup B} &= \left\{(s_A, s_B) \rightarrow \left(s'_A, s'_B\right) |s_A \rightarrow s'_A \in T_A \wedge s_B \rightarrow s'_B \in T_B\right\} \end{aligned}$$

$$\begin{aligned} \boxed{B \oslash A} &= \left(S_A \times S_B, \left(s_0^A, s_0^B\right), \left\{(s_A, s_B) \,|s_B \in F_B\right\}, T_{B \oslash A}\right) \\ T_{B \oslash A} &= \left\{(s_A, s_B) \xrightarrow{1} \left(s'_A, s'_B\right) |s_A \rightarrow s'_A \in T_A \wedge s_B \rightarrow s'_B \in T_B \wedge s_A \in F_A\right\} \cup \\ &\quad \left\{(s_A, s_B) \xrightarrow{\varepsilon} \left(s'_A, s'_B\right) |s_A \rightarrow s'_A \in T_A \wedge s_B \rightarrow s'_B \in T_B \wedge s_A \notin F_A\right\} \end{aligned}$$

$$\begin{aligned} \boxed{A \odot B} &= \left(S_A \times S_B, \left(s_0^A, s_0^B\right), \left\{(s_A, s_B) \,|s_A \in F_A \wedge s_B \in F_B\right\}, T_{A \odot B}\right) \\ T_{A \odot B} &= \left\{(s_A, s_B) \rightarrow \left(s'_A, s'_B\right) |s_A \rightarrow s'_A \in T_A \wedge s_B \rightarrow s'_B \in T_B \wedge s_A \in F_A\right\} \cup \\ &\quad \left\{(s_A, s_B) \rightarrow \left(s'_A, s_B\right) |s_A \rightarrow s'_A \in T_A \wedge s_A \notin F_A\right\} \end{aligned}$$

Correctness of algebraic operations

Theorem 3.1. *For all deterministic firing-time automata A, B:*

1. $(A \cup B)$ *and* $(A \odot B)$ *are also deterministic firing-time automata.*
2. $\emptyset \odot A = A \odot \emptyset = \emptyset$ *and* $\{1\}^* \odot A = A \odot \{1\}^* = A$.
3. $\emptyset \oslash A = \emptyset$ *and* $A \oslash \{1\}^* = A$.
4. *If* $L(A) \supseteq L(B)$ *then*
$$A \odot (B \oslash A) \equiv B$$
5. *As a corollary, from the fact that* $L(A \cup B) \supseteq L(B)$*, we get:*
$$(A \cup B) \odot (B \oslash (A \cup B)) \equiv B$$

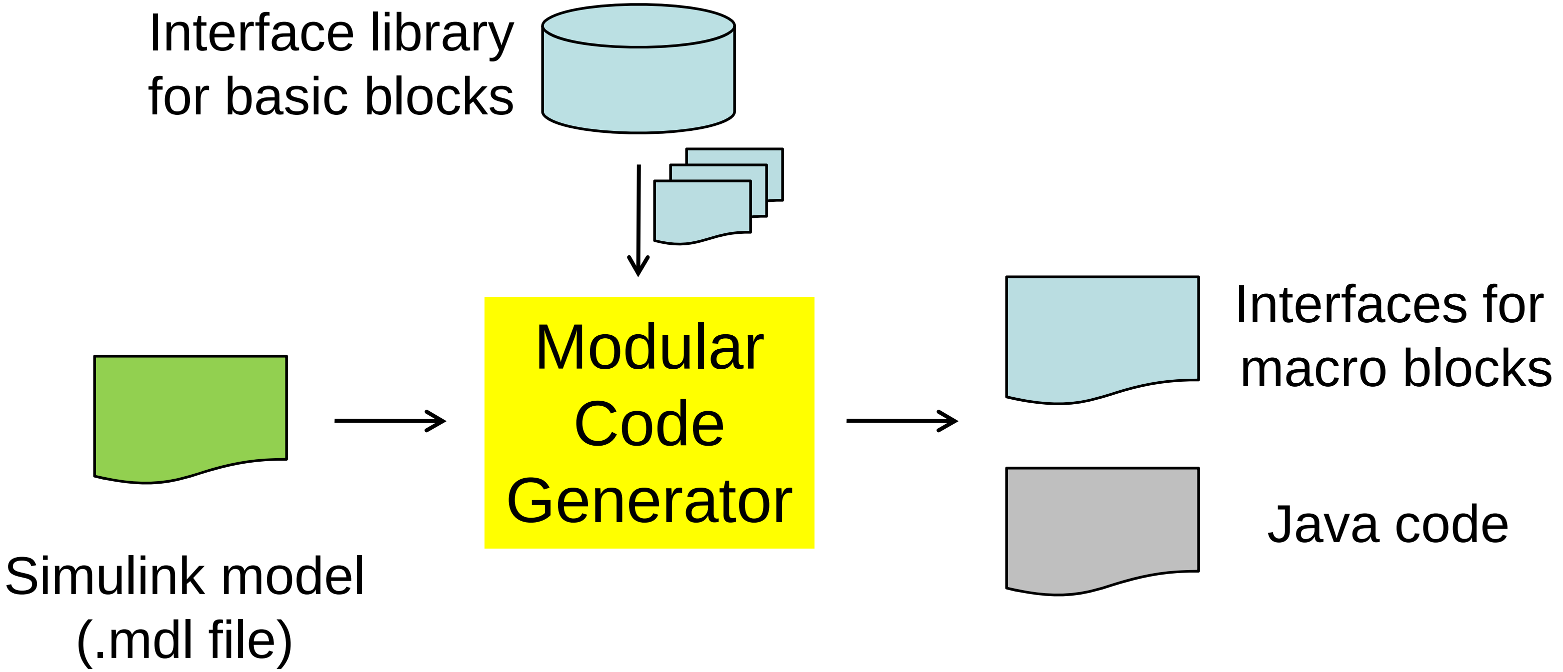
Firing Time Automata: summary

- Closed under union => can represent sets of firing times precisely
- Algebraic manipulation (“product”, “division”)
- Implemented as simple counters + set of accepting states
- Efficient code:
  - Fire a block only when we have to

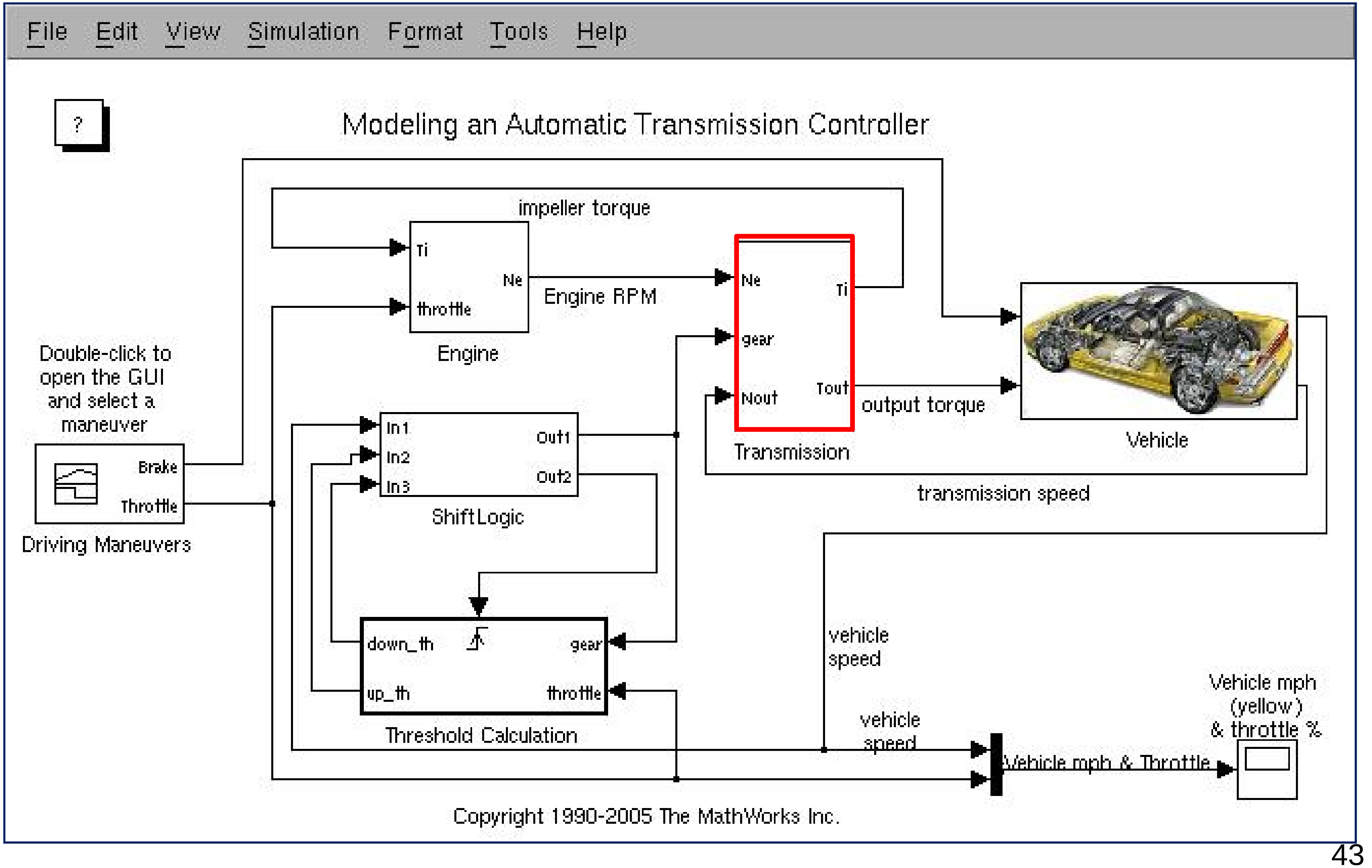
Tool

demo available!

- Implemented in Java by Roberto Lublinerman
- Three clustering methods: “step-get” (max. 2 functions), optimal modularity (over-lapping clusters), and optimal disjoint clustering (uses SAT)



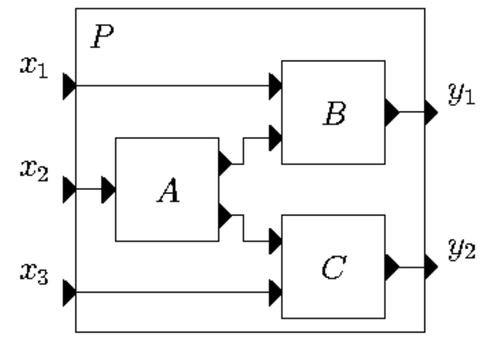
Demo

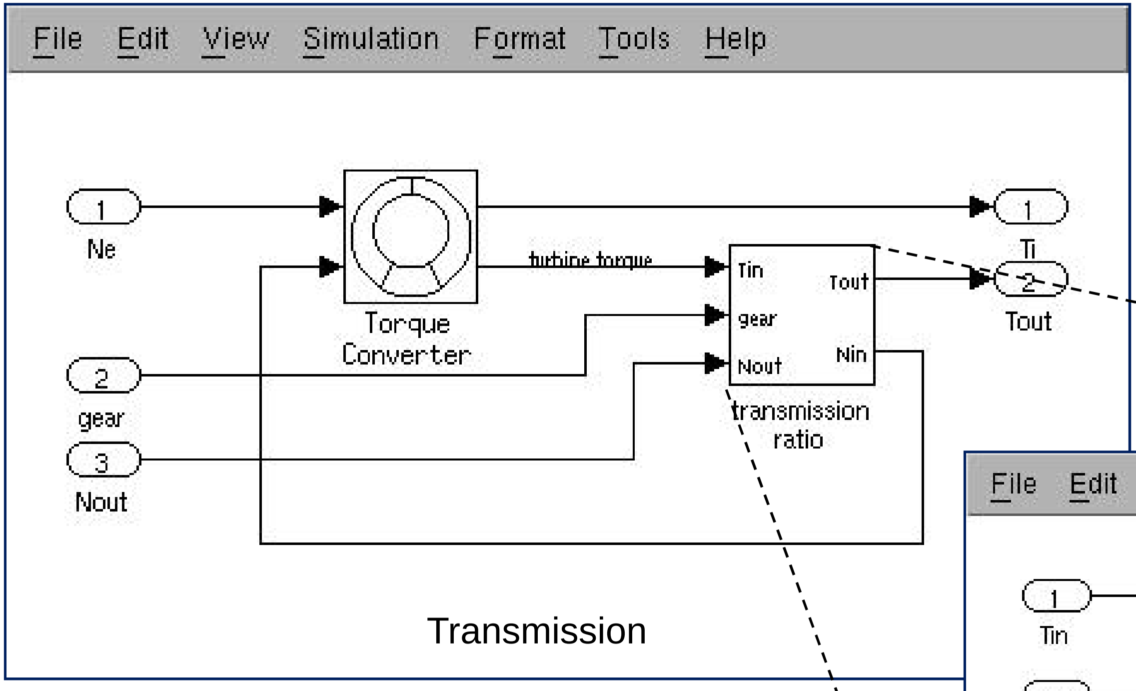


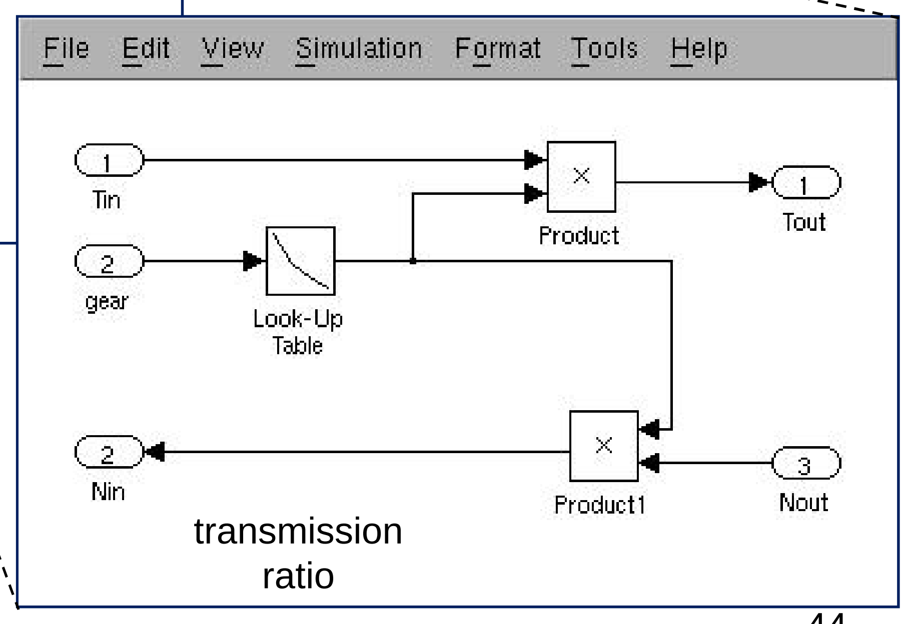
File Edit View Simulation Format Tools Help
?
Modeling an Automatic Transmission Controller
impeller torque
Ti
Ne
throttle
Engine RPM
Engine
Ne
Ti
gear
Nout
Tout
output torque
Transmission
Vehicle
Double-click to open the GUI and select a maneuver
Brake
Throttle
Driving Maneuvers
In1
In2
In3
Out1
Out2
ShiftLogic
transmission speed
down_th
gear
up_th
throttle
Threshold Calculation
vehicle speed
vehicle speed
Vehicle mph (yellow) & throttle %
Vehicle mph & Throttle
Copyright 1990-2005 The MathWorks Inc.

Demo

remember this?







Experiments

- Examples from Simulink's demo suite, plus two from industrial partners

- Experimental results:

model	no. blocks			max no.	max no.	total no. intf. func.			total code size (LOC)			
name	total	macro	C,NS,MS	outputs	sub-blocks	S-G	Dyn	ODC	S-G	Dyn	ODC	max red.
ABS	27	3	1,0,2	1	13	4	4	4	57	57	57	—
Autotrans	42	9	4,0,5	2	11	fails	13	14	fails	108	101	14:6
Climate	65	10	4,0,6	4	29	12	14	14	144	165	144	42:26
Engine1	55	11	2,1,8	2	12	18	18	18	132	140	132	19:11
Engine2	73	13	3,2,8	2	13	20	20	20	180	188	180	19:11
Power window	75	14	6,2,6	3	11	20	21	21	180	199	183	32:16
X1	82	16	2,5,9	3	14	19	19	19	182	182	182	—
X2	112	16	7,9,0	5	14	22	24	24	245	342	261	108:27

- Code reduction up to 75% for some blocks
- Execution time: negligible

Conclusions

- Modular code generation from synchronous models
  - Long-standing problem, sometimes claimed impossible to solve
- General framework, multiple solutions
  - Fundamental trade-offs: modularity, reusability, code size
- Key ideas: abstraction and interfaces
- Optimality results
- Prototype implementation
- Extensions to triggered and timed diagrams
  - Enrich interface with additional information (timing)

Thank you

Questions?

References

- R. Lublinerman and S. Tripakis. *Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams*, DATE'08.
  - http://www-verimag.imag.fr/~tripakis/papers/date08.pdf
- R. Lublinerman and S. Tripakis. *Modular Code Generation from Triggered and Timed Block Diagrams*, RTAS'08
  - http://www-verimag.imag.fr/~tripakis/papers/rtas08.pdf
- R. Lublinerman and S. Tripakis. *Modular Code Generation from Synchronous Block Diagrams – Modularity vs. Code Size*, POPL'09
  - http://www-verimag.imag.fr/~tripakis/papers/popl09.pdf