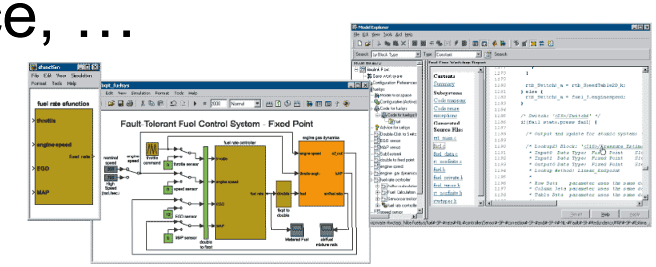
Modular code generation from synchronous block diagrams

Stavros Tripakis – Cadence Research Labs
Roberto Lublinerman – Penn State

ArtistDesign WFCD
Atlanta, Oct '08

Context: embedded software

- High-level **modeling** languages, e.g.:
  - Simulink/Stateflow, SCADE, SystemC, …
- Used for modeling/**simulation**, e.g.:
  - Model discrete-time controller + continuous-time plant in Simulink,
  - Simulate and eye-ball to check stability
- But increasingly also for **code generation**:
  - E.g., Real-Time Workshop, dSpace, …



Code generation from synchronous models – previous work

- Synchronous models:
  - Simulink, synchronous languages (Lustre, Esterel, …), …

- Different execution platforms:
  - Single-processor (“centralized”):
    - Single-thread, no RTOS: classic
    - Multi-thread, preemptive scheduling [ACM TECS ‘08]
  - Multi-processor (“distributed”):
    - Time Triggered Architecture (TTA) [LCTES’03]
    - Asynchronous networks with bounded FIFO queues [IEEE TC ’08]
    - Loosely TTA [IEEE TC ’08]

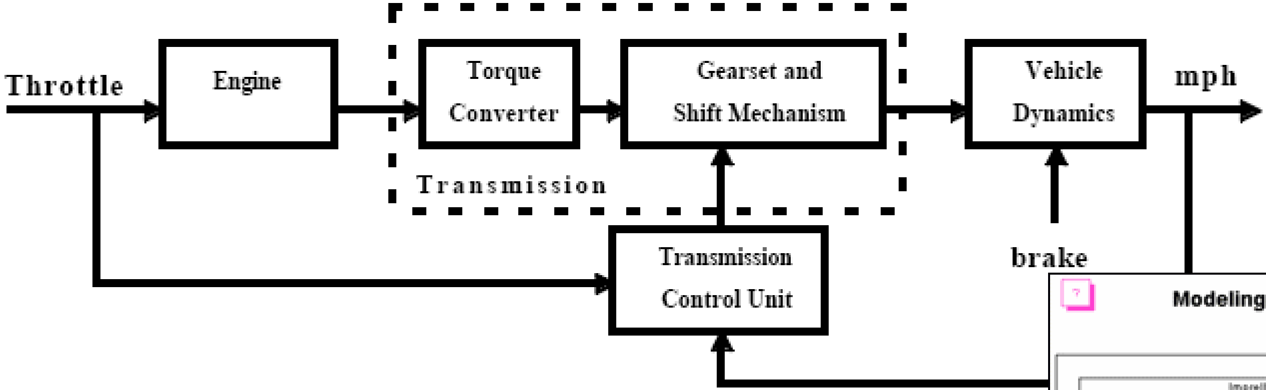
- Different solutions, tailored to each platform

- Focus: preservation of the semantics!

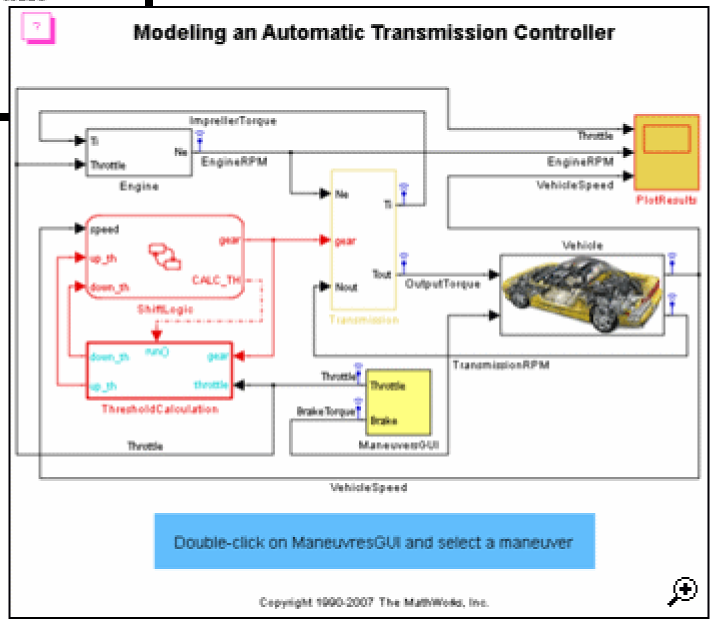
- This talk: **modular** code generation

Synchronous block diagrams

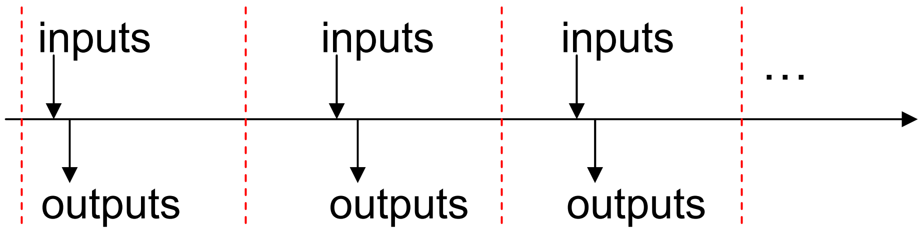
- Fundamental model behind (discrete-time) **Simulink**, or SCADE
- Also very close to synchronous languages: Lustre, Esterel, …



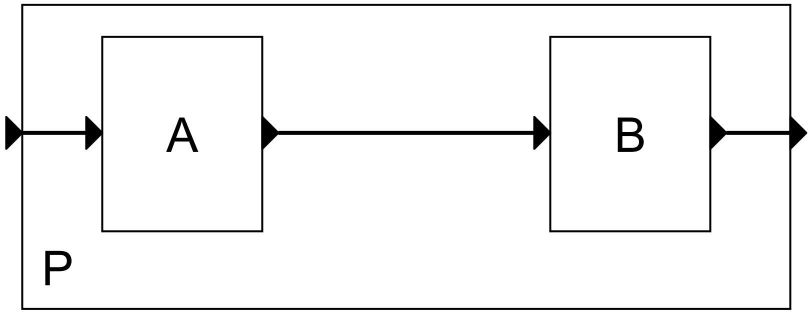
Copyright The Mathworks



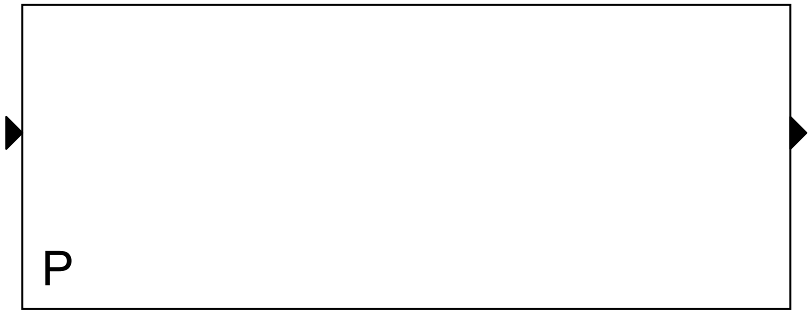
Synchronous, deterministic semantics:



Hierarchy



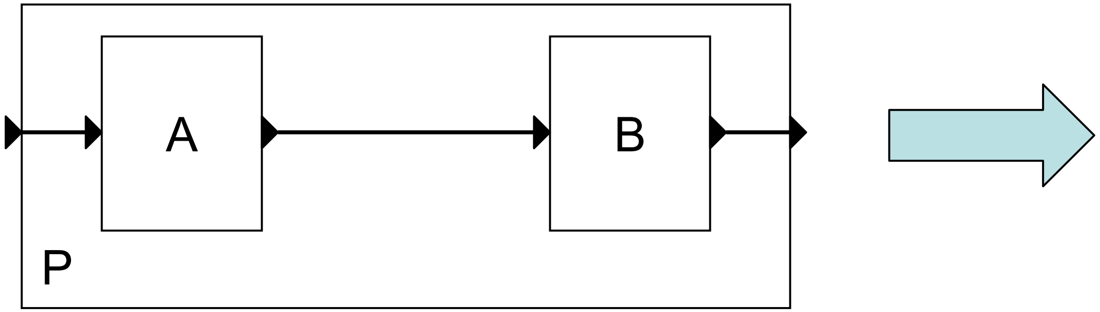
Hierarchy



Fundamental modularity concept

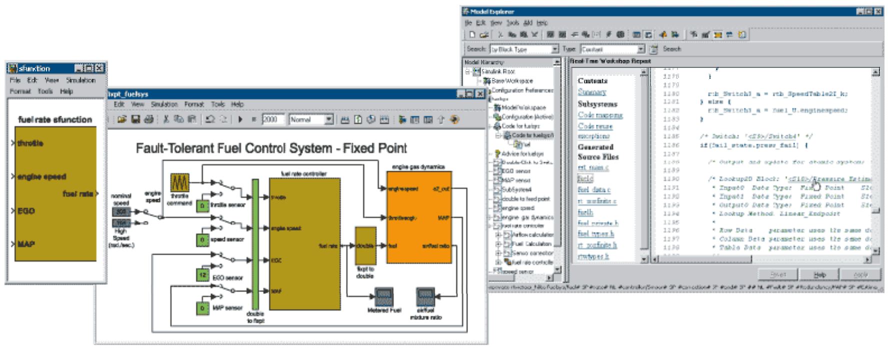
Code generation

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



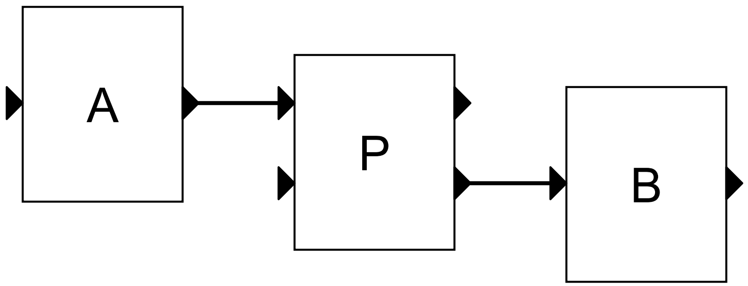

```
P.step( in ) returns out {

    tmp := A.step ( in );
    out  := B.step ( tmp );

    return out;
}
```

- Code may be used for simulation or embedded control
  - Cf. Real-Time Workshop™

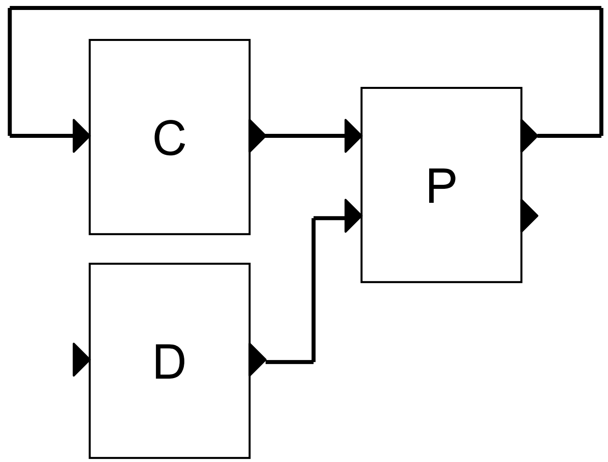


Modular code generation

- Code should be independent from context:



Will P be connected like this?

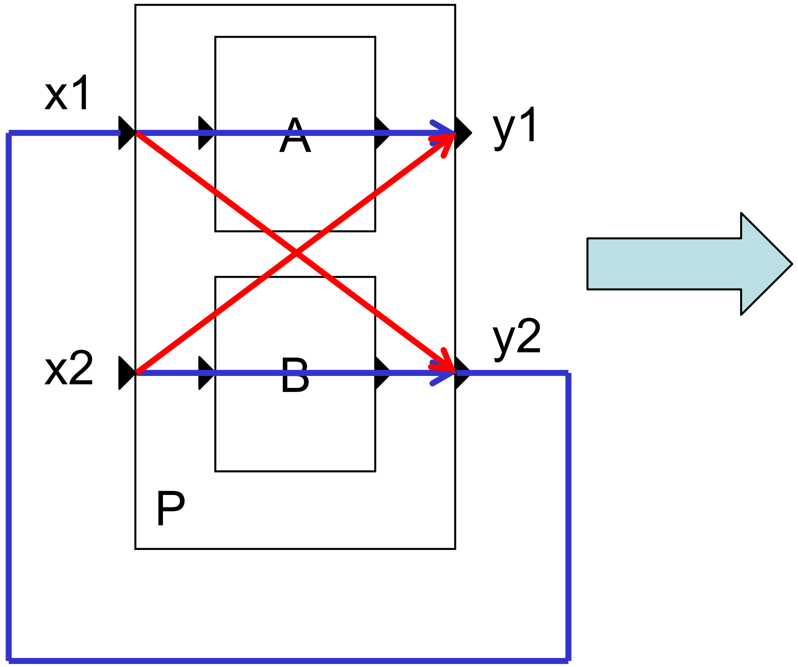


…or like that?

- Enables component-based design
- Takes care of IP issues
- Cf. AUTOSAR

Problem: “monolithic” code

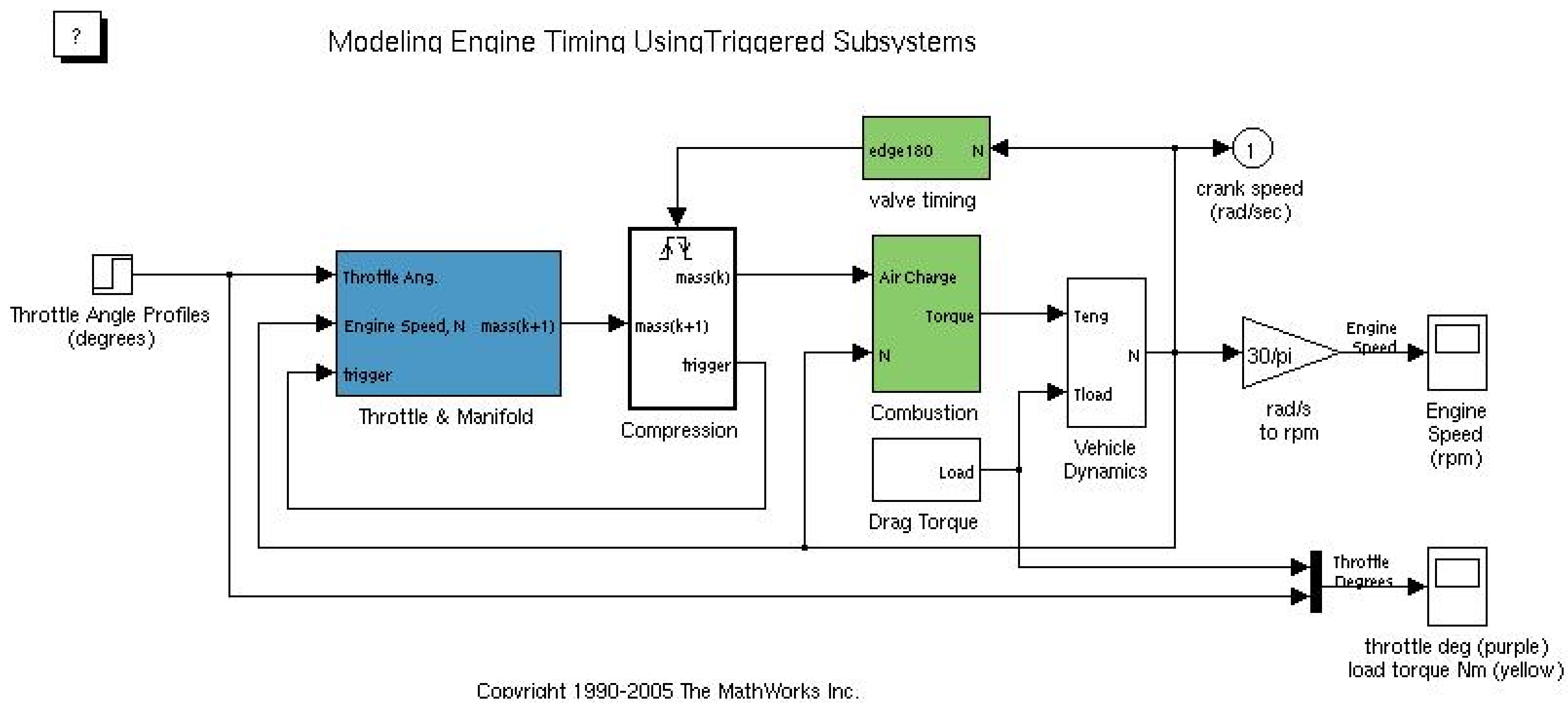
False I/O dependencies




```
P.step( x1, x2 ) returns (y1, y2) {

     y1 := A.step ( x1 );
     y2 := B.step ( x2 );

     return (y1, y2);
}
```

How common is this in practice?

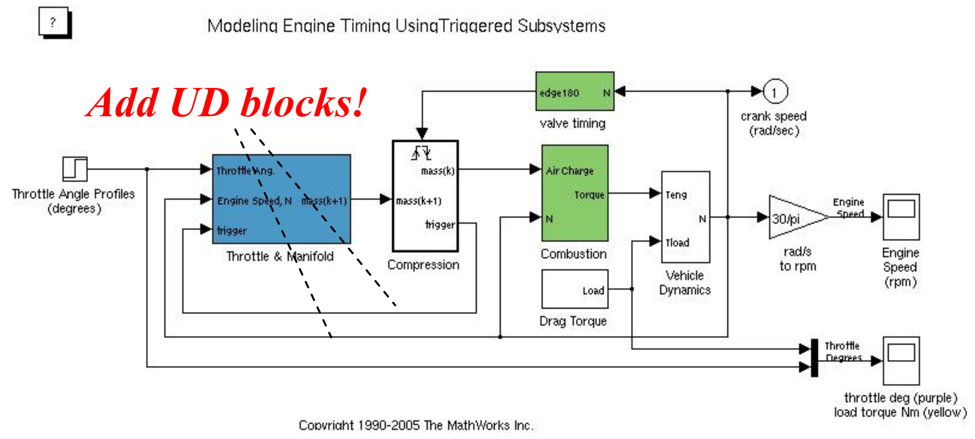
True in all examples we tried



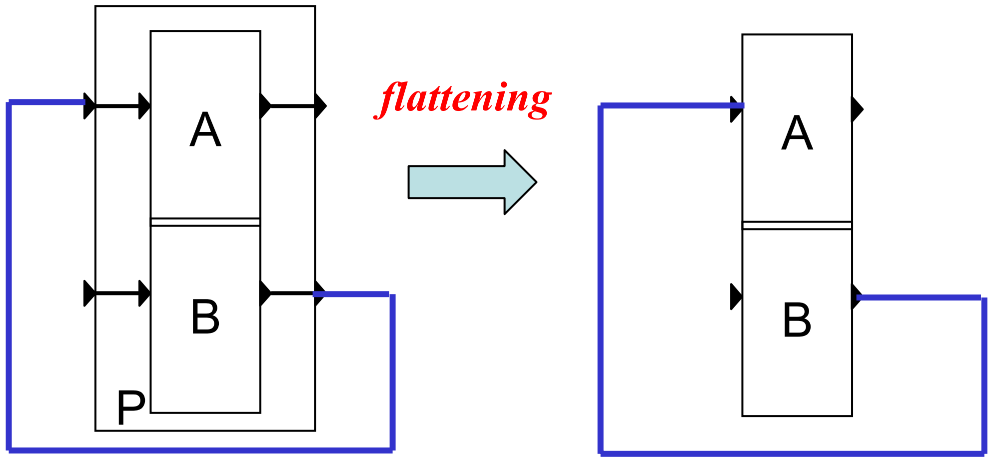
Engine control model in Simulink

Code generation – state of the art

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



- Or flatten (c.f. Simulink)
  - Remove diagram hierarchy
  - Check for dependency cycles
  - If none, generate code
  - Otherwise, reject diagram



- Non-modular!

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
  - Hope for the best:
    - The fix-point may still contain "bottom" values
  - Unacceptable for safety-critical software

- Could check whether diagram is constructive [Malik'94, Berry et al.'96]
  - Expensive
  - Needs semantic information:
    - What is the function that this block computes?
    - Contrary to our black-box view

Our solution [DATE'08, RTAS'08, POPL'09]

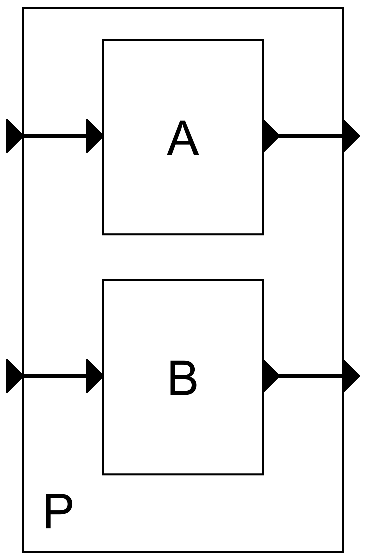
- A general solution to the problem
  - No more flattening
  - No restrictions: handles all diagrams that can be handled by flattening
- A **set** of modular code generation algorithms
  - Some give more modular code than others
  - Notion of modularity is quantified
  - Exposes two fundamental trade-offs:

Modularity vs. Reusability	Modularity vs. Code size

- Optimality results
  - How to generate an optimal (minimal) interface
- Complexity results
  - Some problems are polynomial, some NP-complete

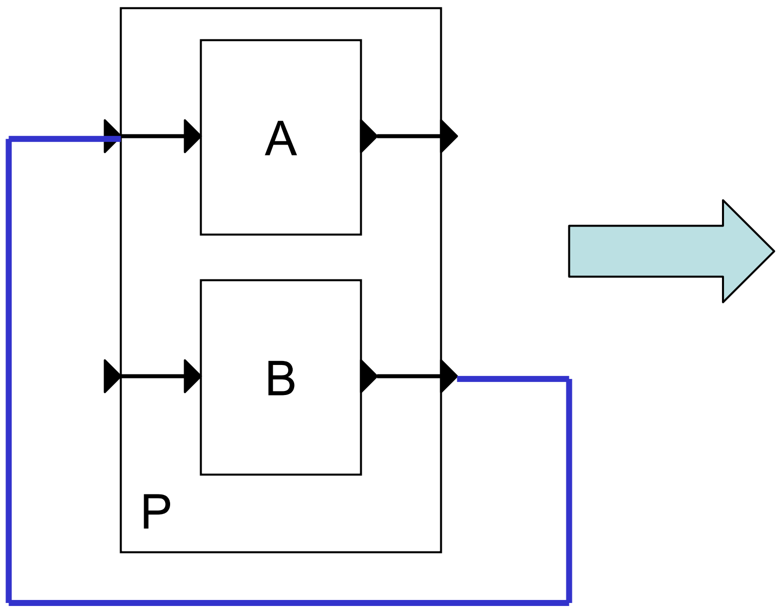
How do we do it?

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



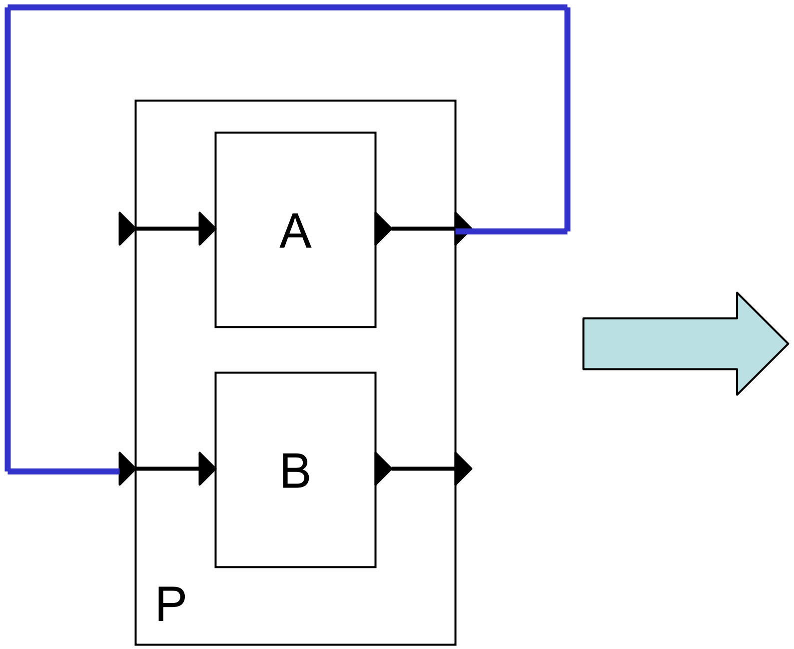

```
class P {
  public P.step1( in1 ) returns out1;
  public P.step2( in2 ) returns out2;

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

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

How do we do it?




```
class P {
  public P.step1( in1 ) returns out1;
  public P.step2( in2 ) returns out2;

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

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

How do we do it?

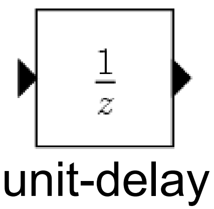



```
class P {
  public P.step1( in1 ) returns out1;
  public P.step2( in2 ) returns out2;

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

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

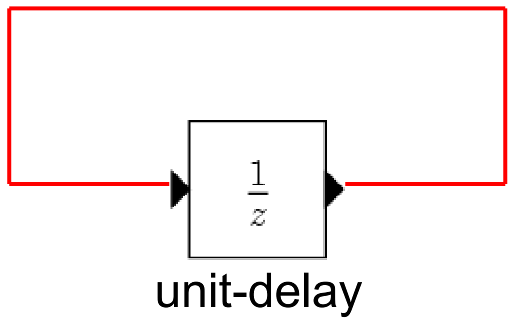
The function call order depends on the usage of the block!

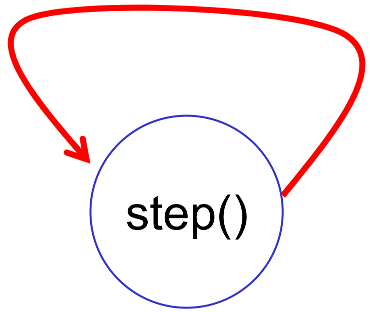
Unit-delay blocks

- Memory element (register):



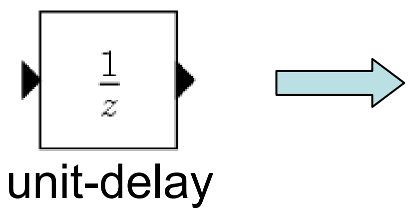
- One interface function is not enough:



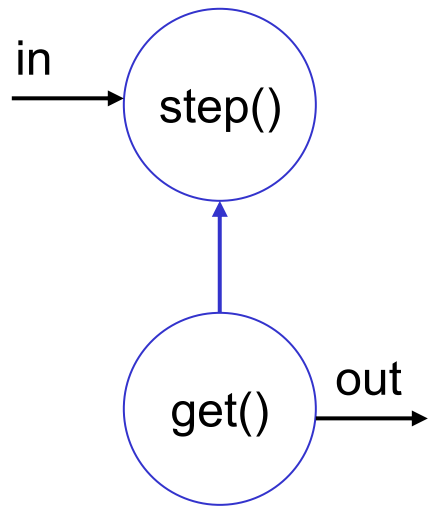


Profile dependency graphs

- Profile also includes a DEPENDENCY GRAPH
- Encodes interface usage constraints

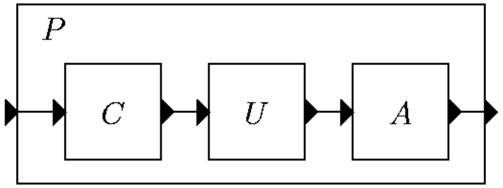



```
class UnitDelay {
  public step( in ) returns void;
  public get(  ) returns out;
  private state;

  step( in ) {
     state := in;
  }

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

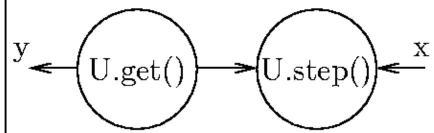
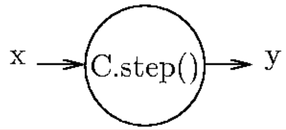


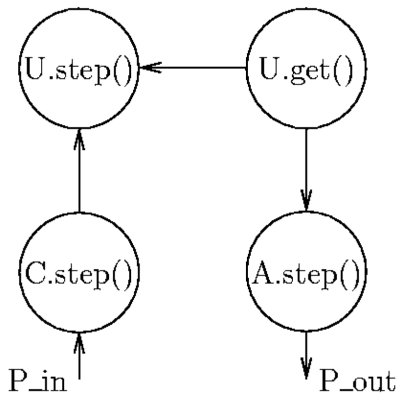
PROFILE
DEPENDENCY
GRAPH

Overall method: example

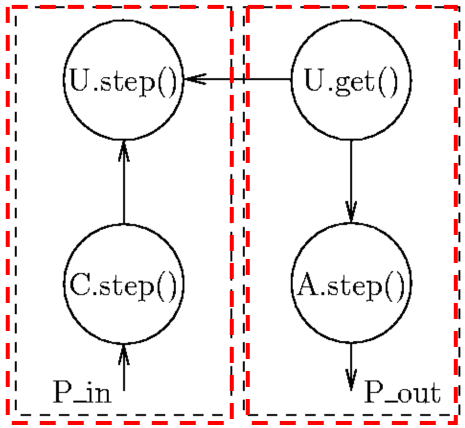


A macro block P and its internal diagram

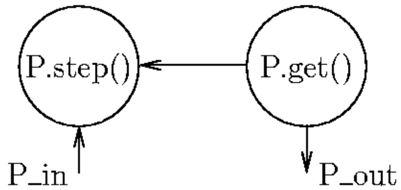
	Interface functions	Profile dependency graphs
Profile of A: (combinational)	A.step(x) returns y;	
Profile of U: (Moore-sequential)	U.get() returns y; U.step(x) returns void;	
Profile of C: (combinational)	C.step(x) returns y;	



SDG of P



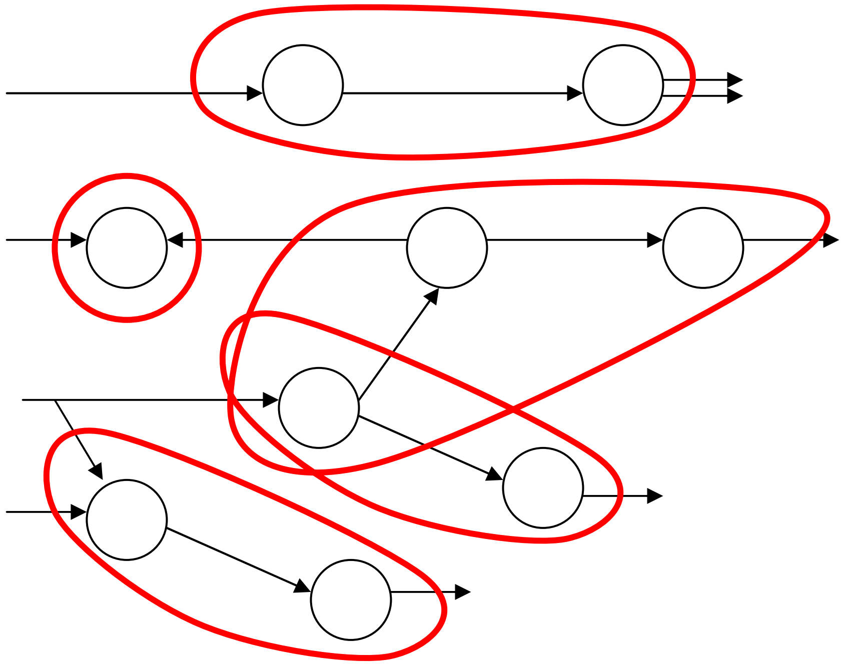
SDG of P clustered in two sub-graphs

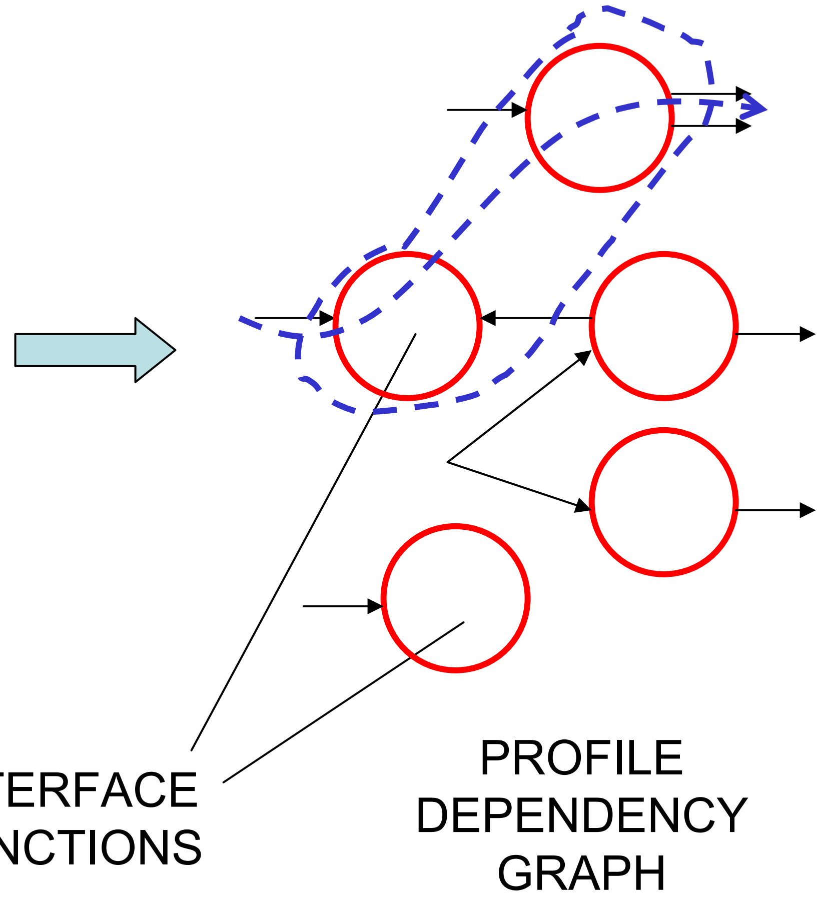


Resulting interface functions and PDG of P

SDGs and clustering

Scheduling Dependency Graph (SDG)
=
composition of PDGs of sub-blocks





different clusterings
=
different tradeoffs

Trade-off:
modularity vs. reusability

[DATE'08]

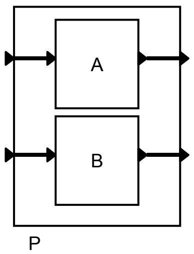
modularity becomes quantifiable

more modular | more reusable

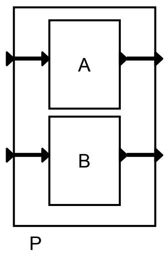
If block has N outputs then maximal reusability can be achieved with <= N+1 functions (tight)

less interface functions | more interface functions

Modularity-optimal method to achieve maximal reusability




```
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 );
  }
}
```

An abstraction-oriented view

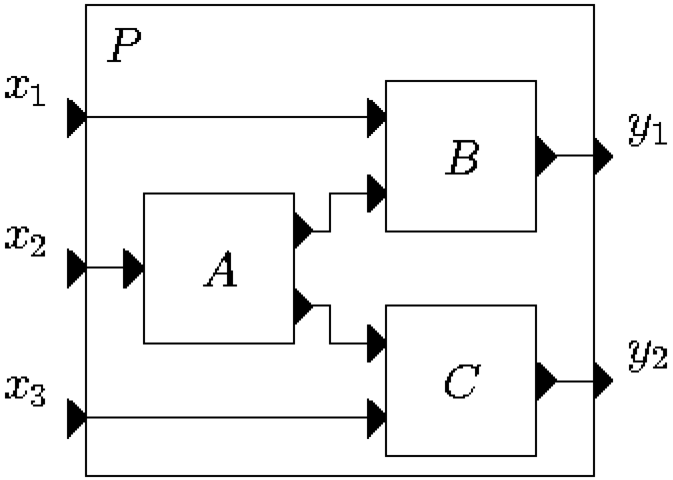
- Interface = an **abstraction** of the block

- It is a **conservative** abstraction
  - All original I/O dependencies are kept
  - More dependencies may be added
  - If no cycle occurs when using the interface, then no cycle would occur if instead we had flattened the block

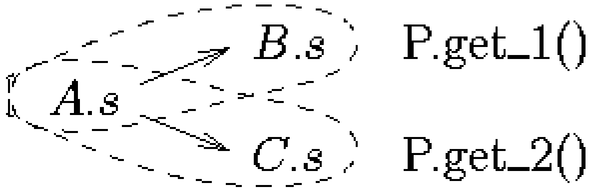
- The **most conservative** abstraction is 1 function:
  - "step" function: computes outputs and updates state
  - Every output depends on all inputs

- An **exact** abstraction always exists
  - The set of I/O dependencies is finite

How to achieve optimality: overlapping clusters



A macro block P



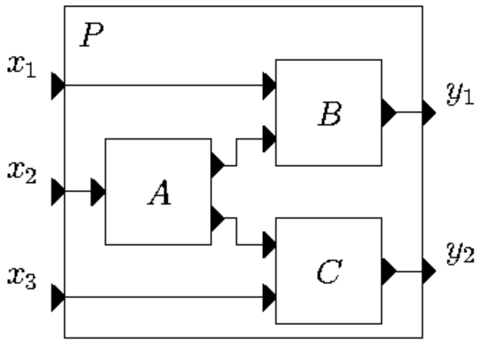
Clustered SDG of P and corresponding interface functions

2 outputs, 2 interface functions (optimal)

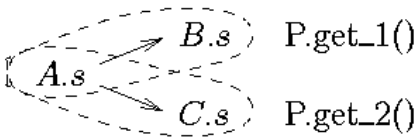
Overlapping clusters => code replication

```
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;
}
```



A macro block P



Clustered SDG of P and corresponding interface functions

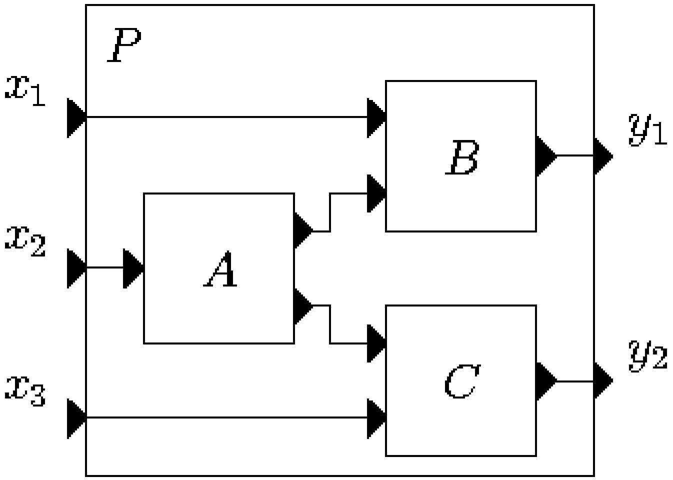
Overlapping clusters => code replication

```
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;
}
```

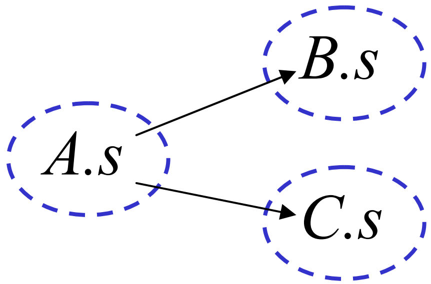
Another trade-off: modularity vs. code size

[POPL'09]



A macro block P

minimize code size =>
non-overlapping (disjoint) clustering



2 outputs, 3 interface functions:
- non-optimal in general
- optimal for disjoint clustering

Optimal disjoint clustering: NP-complete

But it can be reduced to sequence of SAT problems: efficient in practice

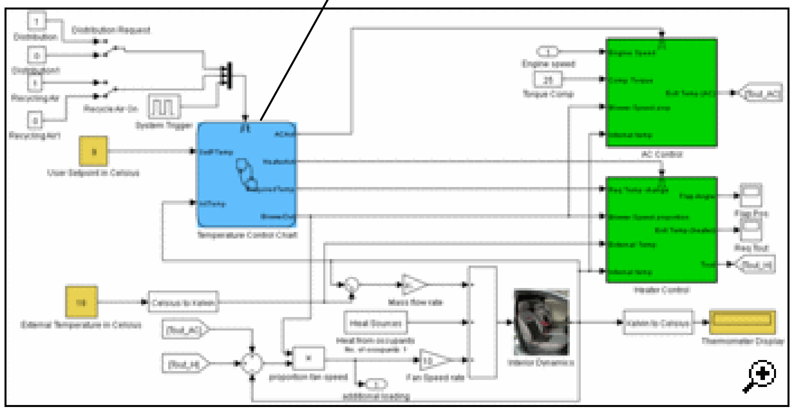
With Christian Szegedy

Extension to triggered and timed diagrams

[RTAS'08]

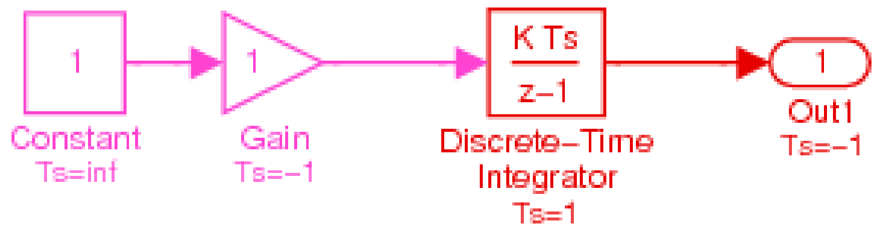
- Triggers and time:
  - Both concepts found in Simulink, SCADE, synchronous languages, …

Triggered block



Simulink/Stateflow diagram

Inline Parameters = on

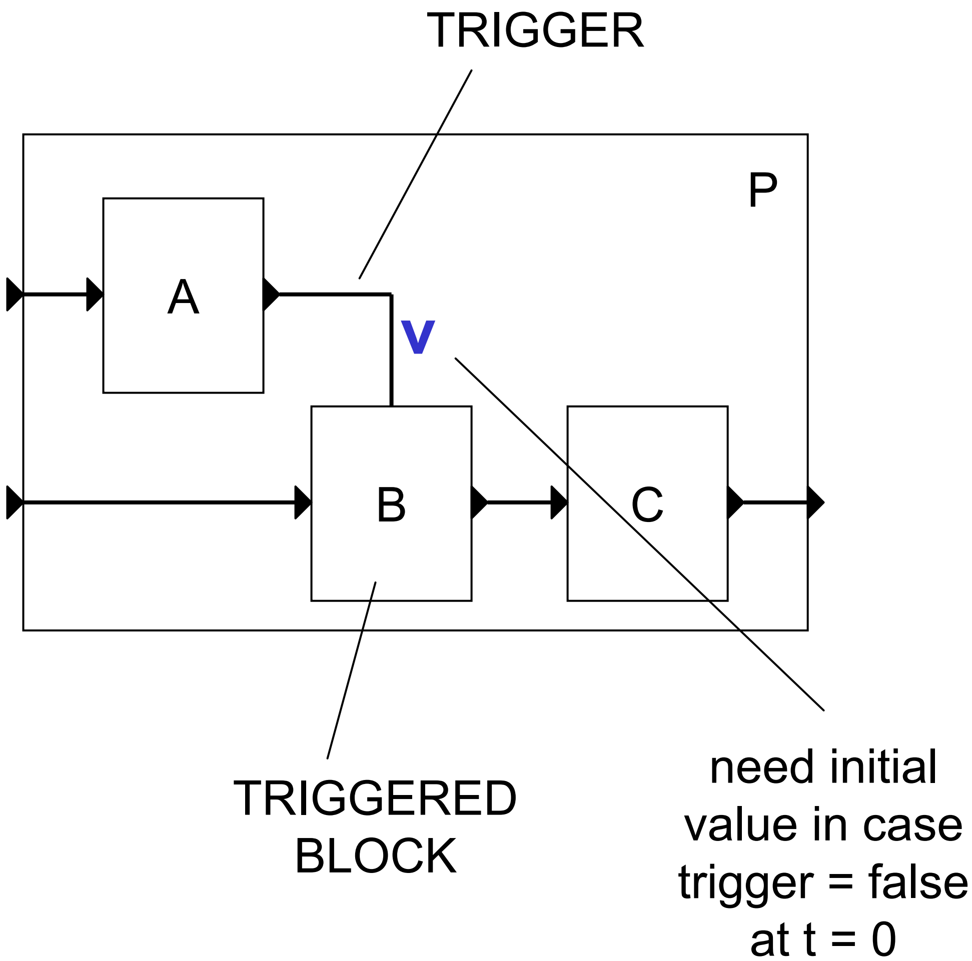


Sample time

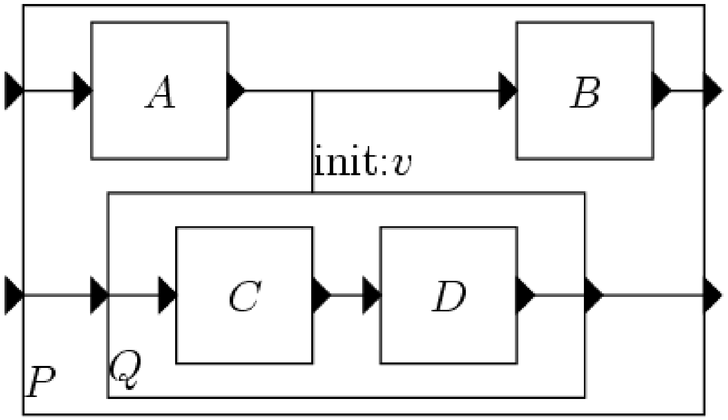
Triggered diagrams

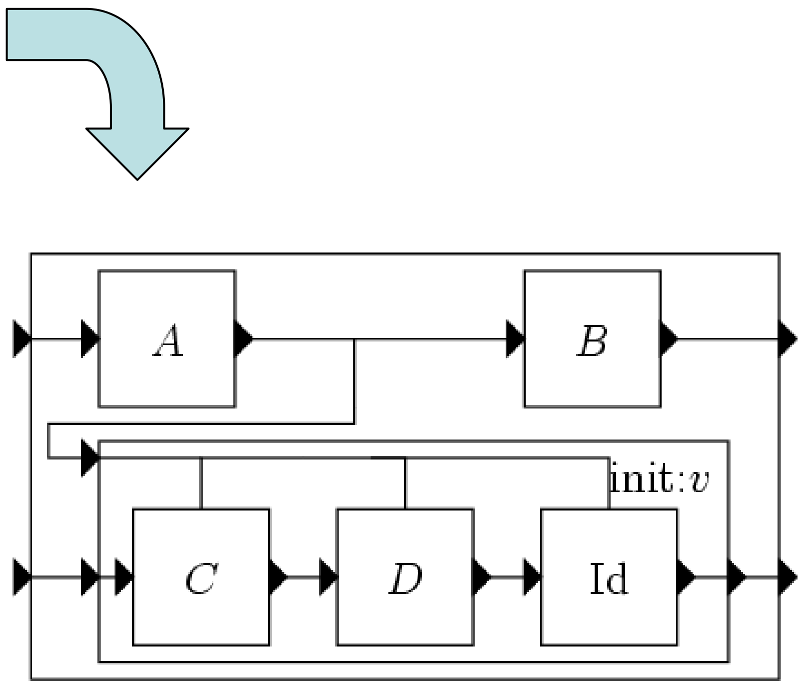
multi-rate models:

- 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

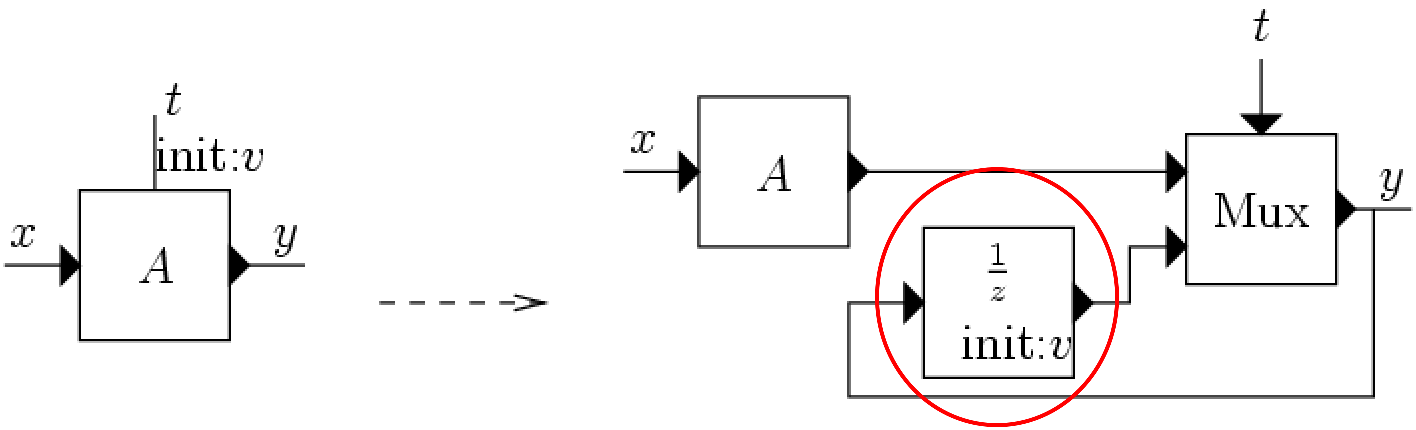


Trigger elimination

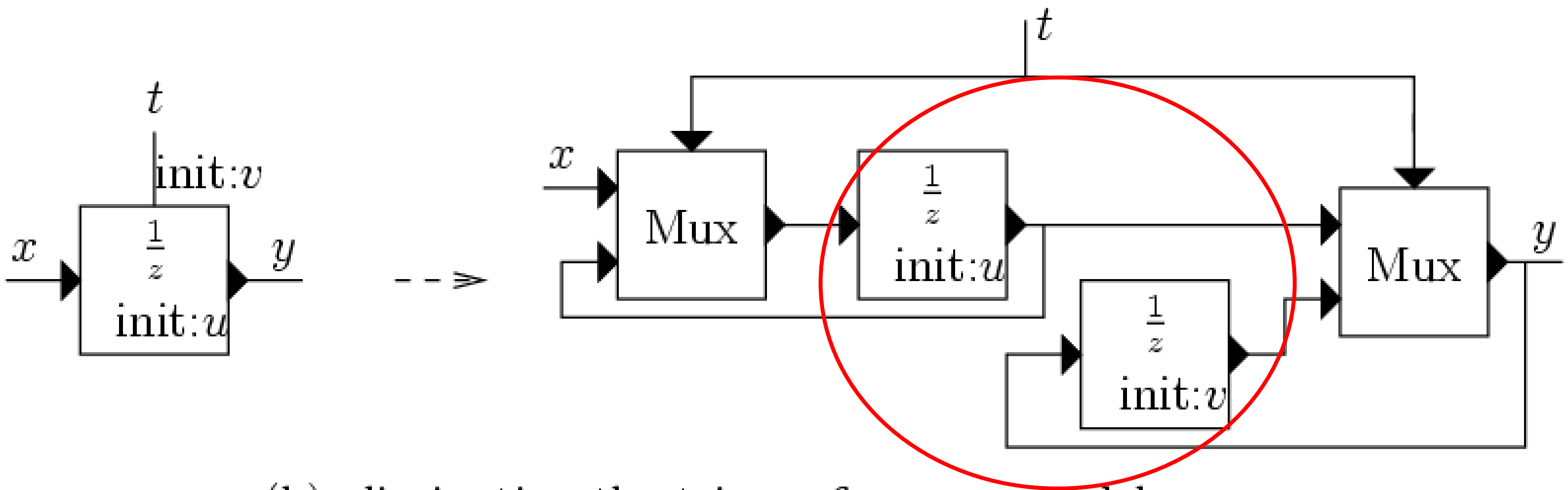




Trigger elimination: atomic blocks



(a) eliminating the trigger from a combinational atomic block

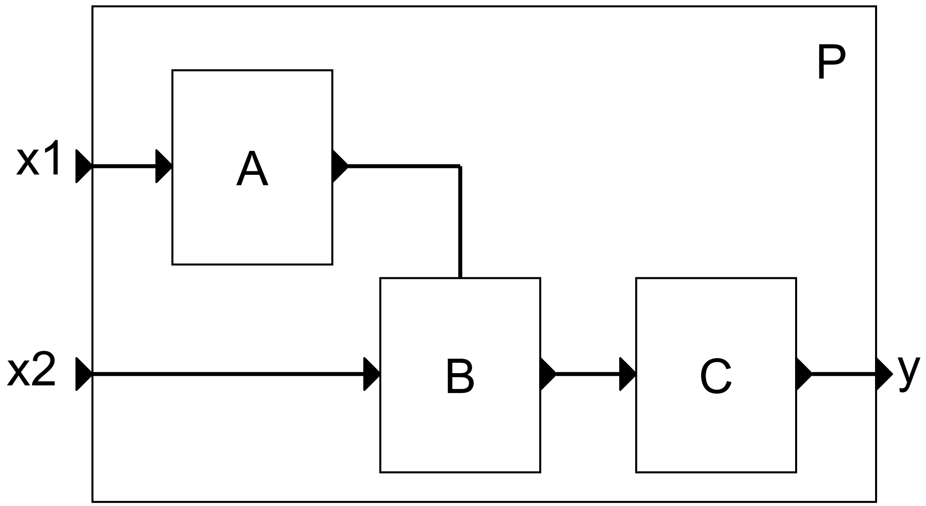


(b) eliminating the trigger from a unit-delay

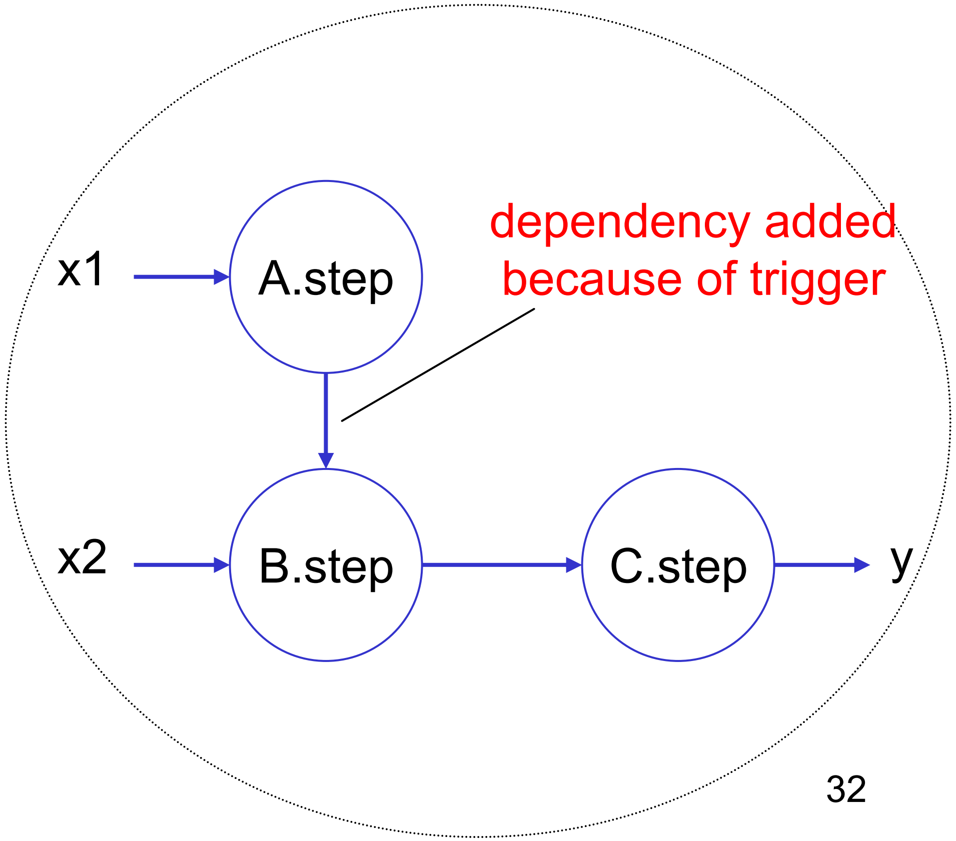
Trigger elimination: summary

- Can be done, efficiently
- But it destroys modularity:
  - Must propagate triggers top-down => “open the boxes”
- Solution:
  - Handle triggers directly, without eliminating them

Handling triggered diagrams directly

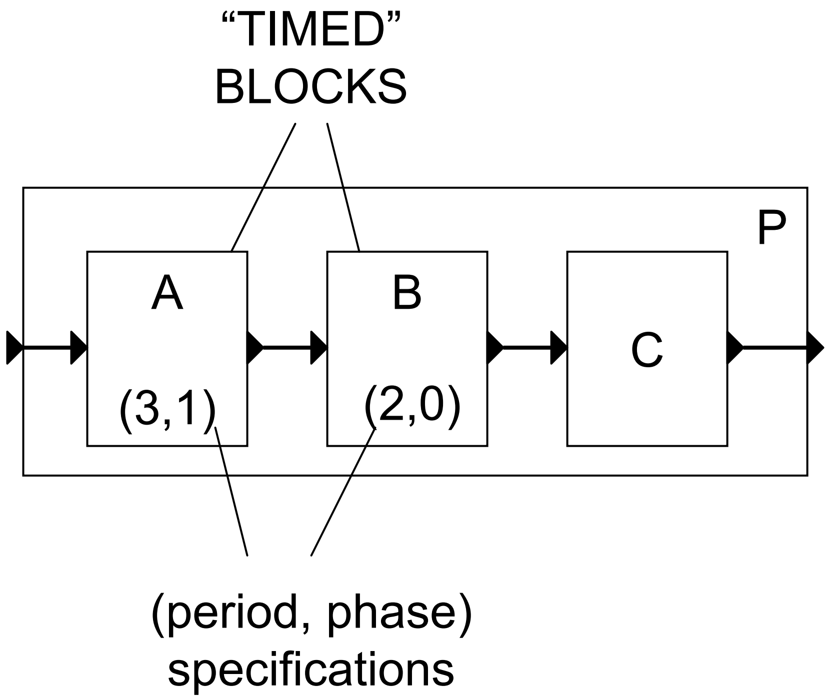


Scheduling Dependency Graph of P:

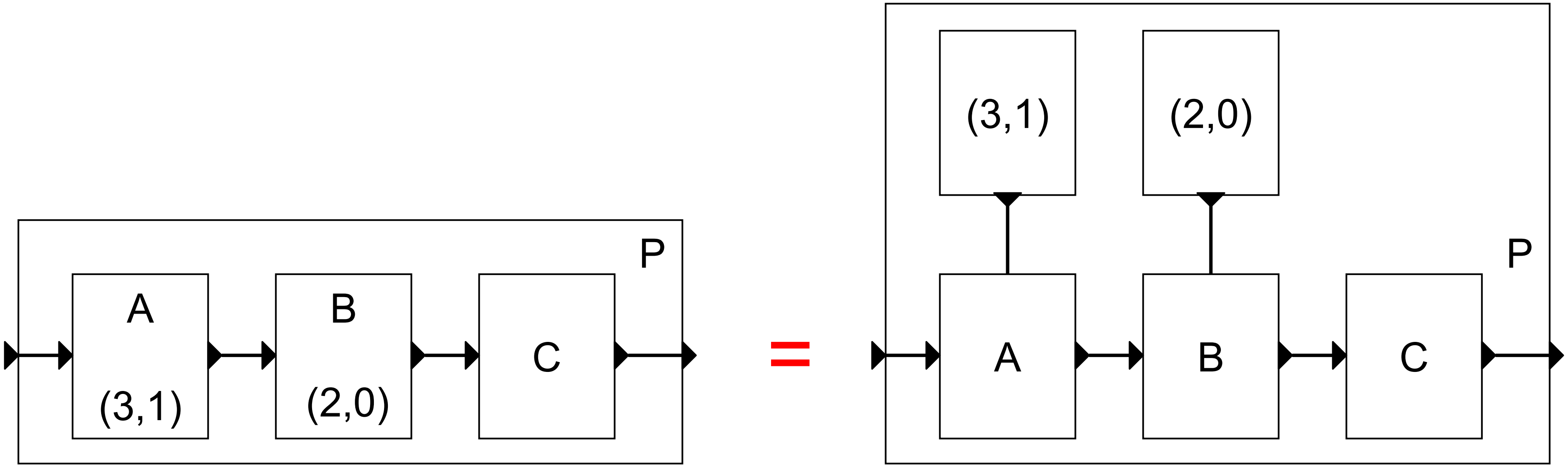


Timed diagrams

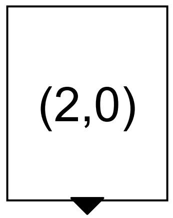
"static" multi-rate models



Timed diagrams = "static" triggered diagrams



where

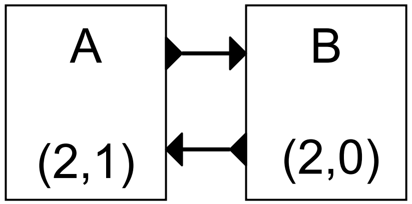


produces: true, false, true, false, …

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

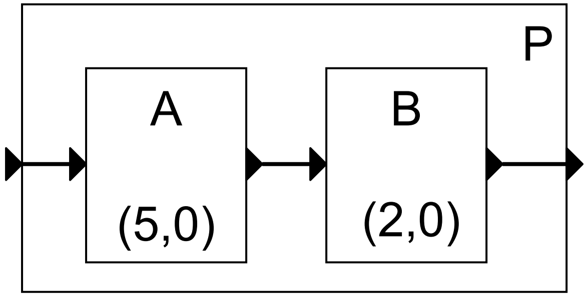


A and B are never active at the same time

=>

Both dependencies are false

Eliminating redundant firings



Q: how often should P be fired?

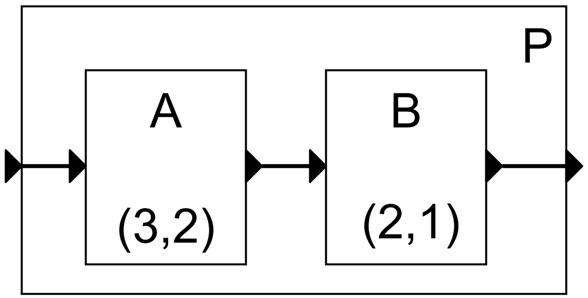
Simple answer: every GCD(5,2) = 1 time unit = at every "clock cycle"

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

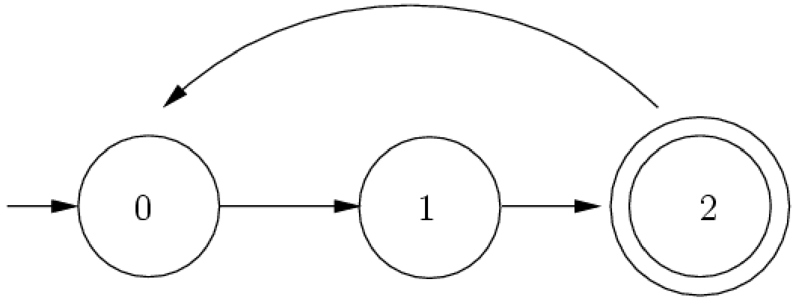
Problem: (period,phase) representation not closed under union

Solution: Firing Time Automata

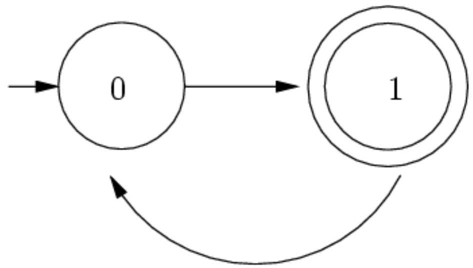
Firing Time Automata

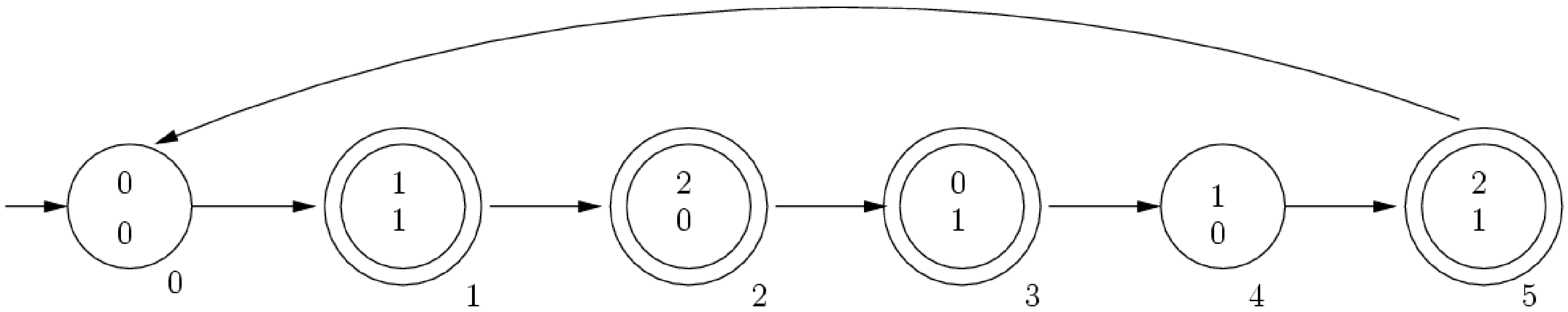


A



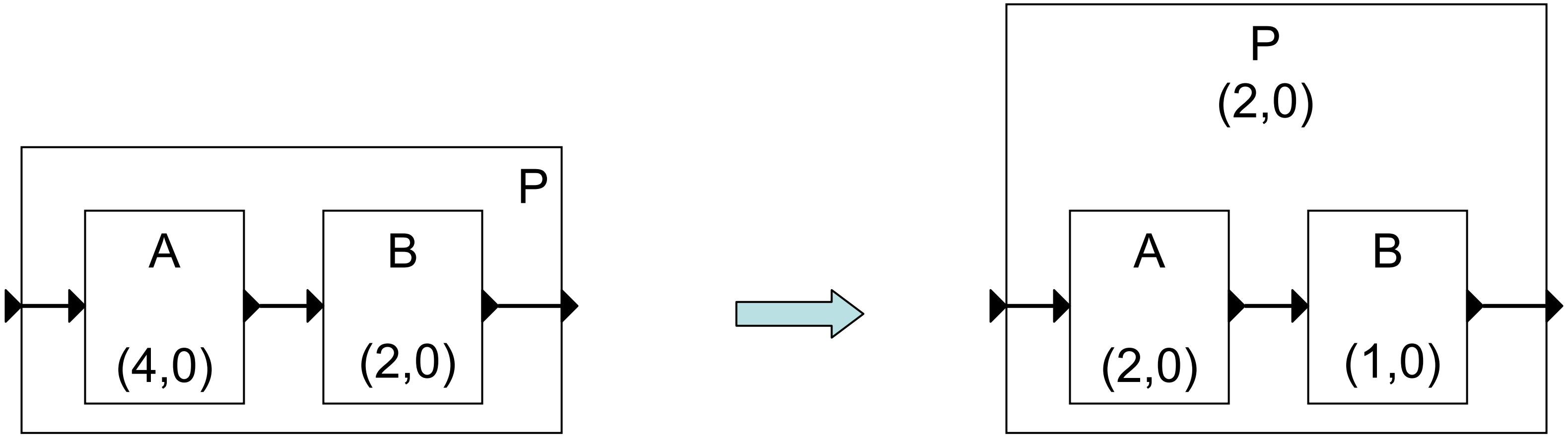
B

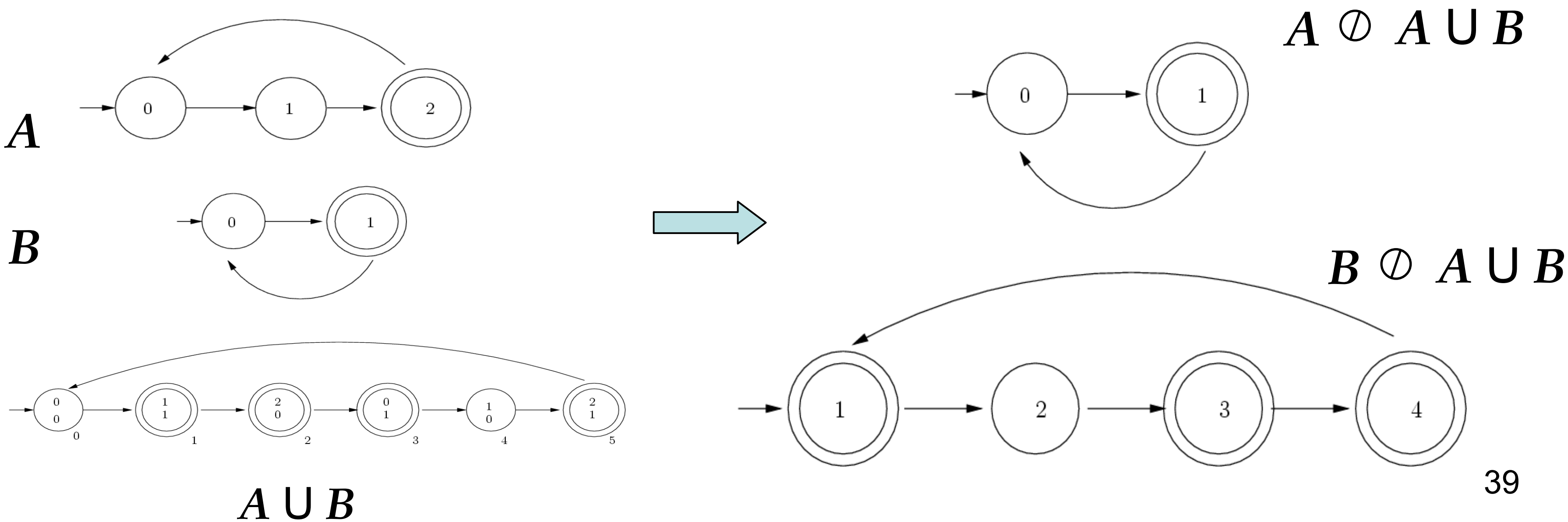




A ∪ *B*

FTA division and multiplication





Firing Time Automata Operations

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

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

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

Firing time automata

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

[POPL'09]

Tool and experiments

- Tool implemented in Java
- Three clustering methods:
  - "step-get": 1 or 2 clusters
  - "dynamic": minimum no. clusters with overlapping
  - "ODC": optimal disjoint clustering (uses SAT solving)
- 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

It's for real!

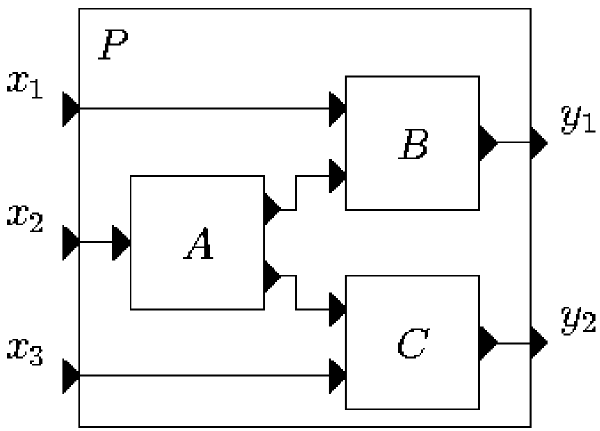


Diagram in our DATE'08 paper

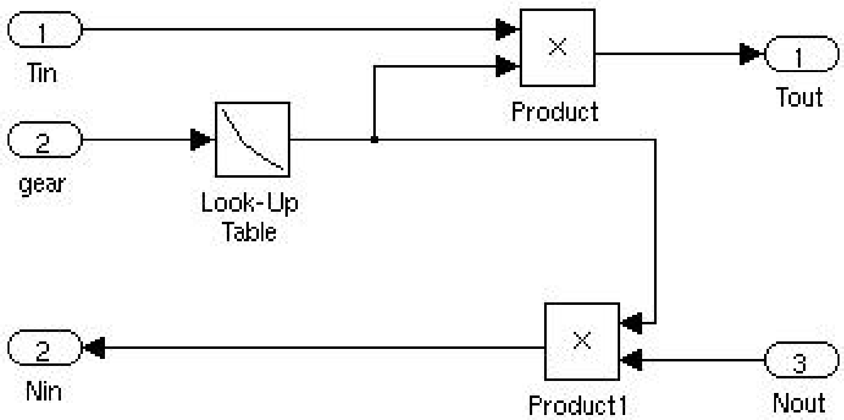


Diagram in one of Simulink demos
(engine control)

They are isomorphic!

Conclusions

- Modular code generation framework
  - No more flattening, no more IP issues, no restrictions on input
  - Handles triggered and timed diagrams
- Spectrum of methods
- Exposed fundamental trade-offs:
  - modularity vs. reusability
  - modularity vs. code size
- Optimality and complexity results
- Prototype tool and experiments

Thank you

- Questions ?