Embedded software implementation: a model-based approach

Stavros Tripakis
Cadence Research Labs
tripakis@cadence.com

Lecture at EE249, Oct 2007

What are these?

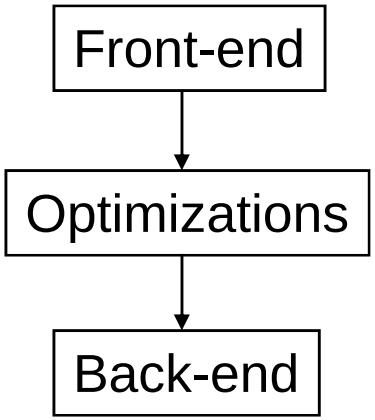
- Alpha
- ARM
- Atmel AVR
- Blackfin
- HC12
- H8/300
- IA-32 (x86)
- x86-64
- IA-64
- MorphoSys
- Motorola 68000
- MIPS
- PA-RISC
- PDP-11
- PowerPC
- R8C/M16C/M32C
- System/390/zSeries
- SuperH
- SPARC
- VAX

- A29K
- ARC
- C4x
- ETRAX CRIS
- D30V
- DSP16xx
- FR-30
- FR-V
- Intel i960
- IP2000
- M32R
- 68HC11
- MCORE
- MMIX
- MN10200
- MN10300
- Motorola 88000
- NS32K
- ROMP
- Stormy16
- V850
- Xtensa
- AVR32

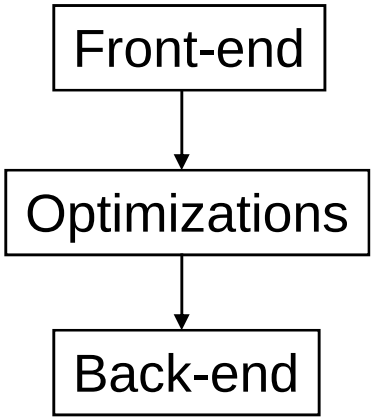
- D10V
- MeP
- MicroBlaze
- Nios II and Nios
- PDP-10
- MSP430
- Z8000

GCC 4.1 target processors [Wikipedia]

What is this?



Structure of GCC [Wikipedia]



Front ends [edit]

Frontends vary internally, having to produce trees that can be handled by the backend. The parsers are hand-coded recursive descent parsers.

Until recently, the tree representation of the program was not fully independent of the processor being targeted. Confusingly, the meaning of a tree was somewhat different for different language front-ends, and front-ends could provide their own tree codes.

In 2005, two new forms of language-independent trees were introduced. These new tree formats are called GENERIC and GIMPLE. Parsing is now done by creating temporary language-dependent trees, and converting them to GENERIC. The so-called "gimplifier" then lowers this more complex form into the simpler SSA-based GIMPLE form which is the common language for a large number of new powerful language- and architecture-independent global (function scope) optimizations.

Optimization [edit]

Optimization on trees does not generally fit into what most compiler developers would consider a front end task, as it is not language dependent and does not involve parsing. GCC developers have given this part of the compiler the somewhat contradictory name the "middle end." These optimizations include dead code elimination, partial redundancy elimination, global value numbering, sparse conditional constant propagation, and scalar replacement of aggregates. Array dependence based optimizations such as automatic vectorization are currently being developed.

Each of the language compilers is a separate program that takes in source code and produces assembly language. All have a common internal structure. A per-language front end parses the source code in that language and produces an abstract syntax tree ("tree" for short), and a back end converts the trees to GCC's Register Transfer Language (RTL). Compiler optimizations and static code analysis techniques (such as FORTIFY_SOURCE[2], a compiler directive which attempts to discover some buffer overflows) are applied to the code. Finally, assembly language is produced using architecture-specific pattern matching originally based on an algorithm of Jack Davidson and Chris Fraser.

Design (programming) vs. implementation (compilation)

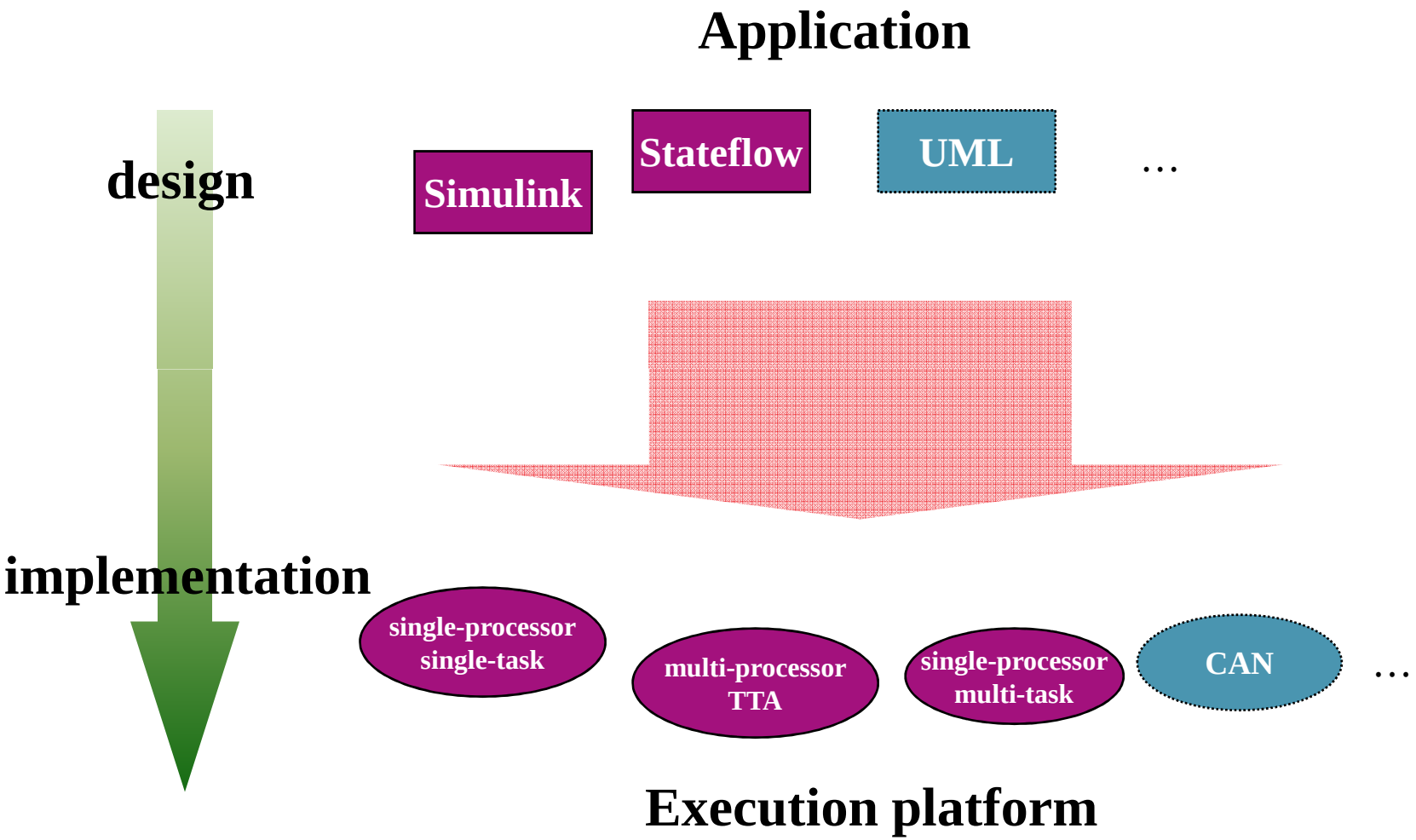
- Design:
  - Focus on function: what should the program do and how to do it
  - Focus on correctness: debugging does/should not depend on whether program runs on Linux/Windows, AMD/Intel
  - Focus on readability, extensibility, maintainability, …: others will come after you to work on this program
  - (Try to) avoid thinking about how to make it run fast (except high-level decisions, e.g., which data-structure to use)

- Implementation:
  - Focus on how to execute the program correctly on a given architecture: compiler knows the instruction set, programmer does not
  - Make it run fast: compiler knows some of that

Embedded software

- This is more or less true for “general-purpose” software…
- …but certainly not for embedded software!
  - Lots of worries due to limited resources: small memory, little power, short time to complete task, …
- Will it ever be?
- We believe so.

Model based design: what and why?

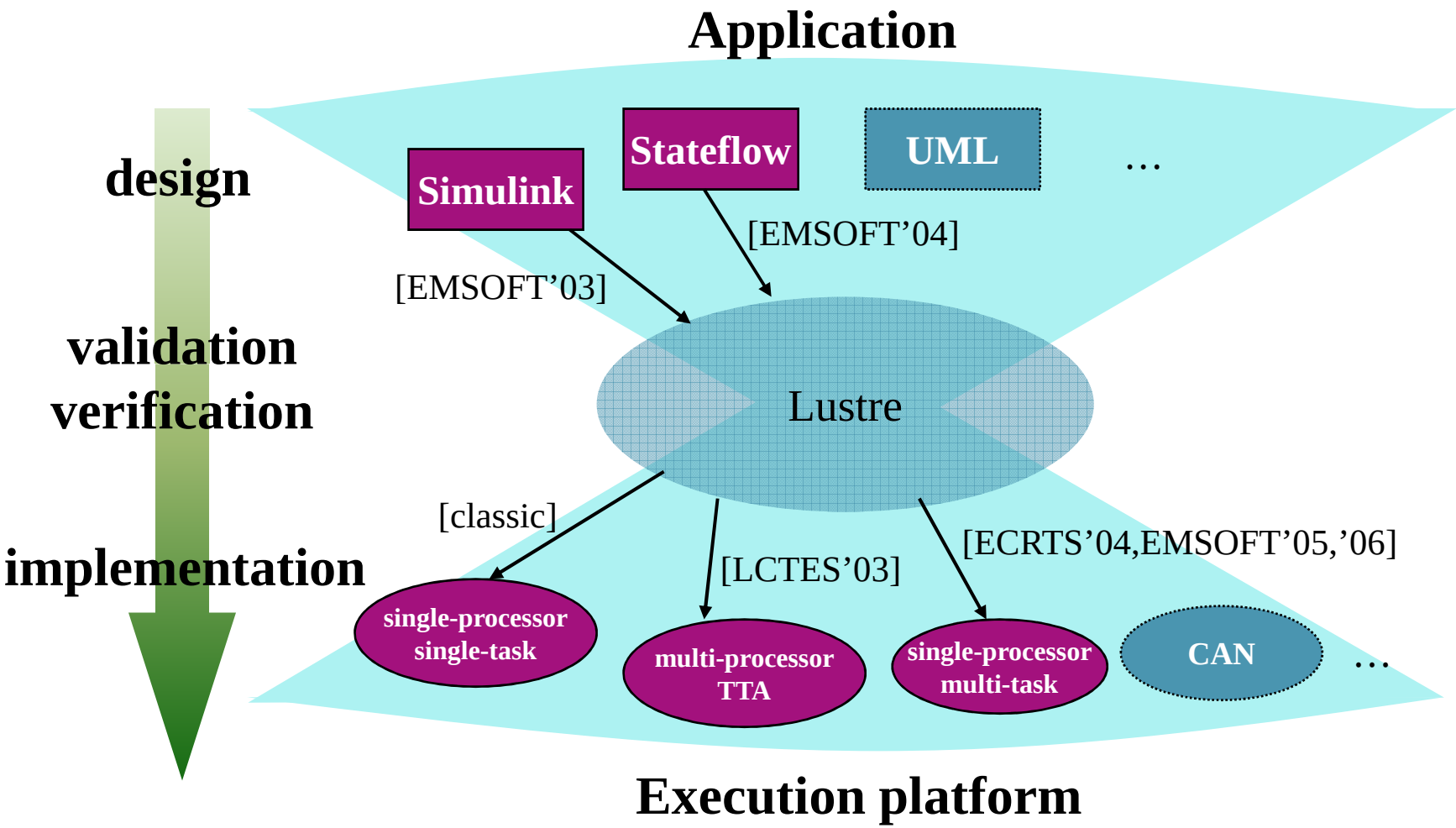


Model based design: benefits and challenges

- Benefits:
  - Increase level of abstraction => ease of design
  - Abstract from implementation details => platform-independence
  - Earlier verification => bugs cheaper to fix
  - Design space exploration (at the “algorithmic” level)
  - Consistent with history (e.g., of programming languages)
- Challenges:
  - High-level languages include powerful features, e.g.,
    - Concurrency, synchronous (“0-time”) computation/communication,…
  - How to implement these features?
    - Do we even have to?

Model based design – the Verimag approach

(joint work with P. Caspi, C. Sofronis, A. Curic, A. Maignan, at Verimag)



Agenda

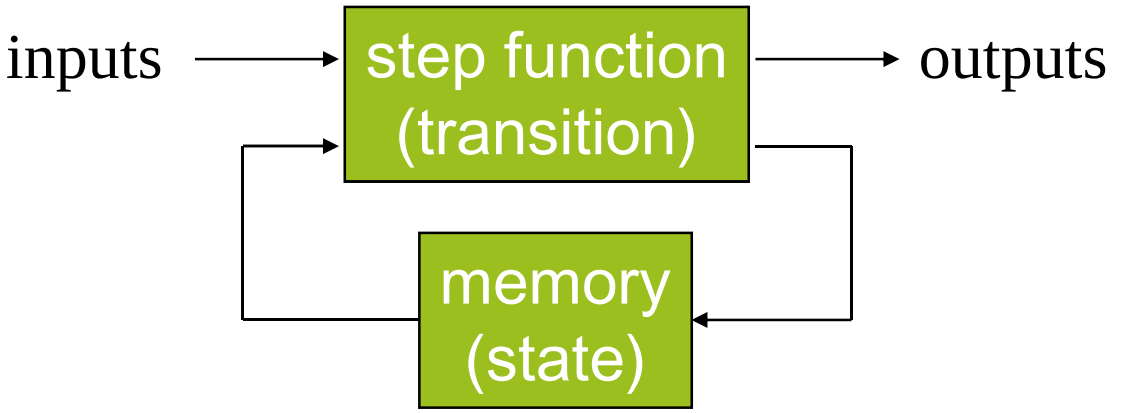
- Part I – from synchronous models to implementations
  - Single-processor/single-task code generation
  - Multi-task code generation:
    - the Real-Time Workshop™ solution
    - a general solution
  - Implementation on a distributed platform:
    - General concerns
    - Implementation on a Kahn process network
    - Implementation on the Time Triggered Architecture
- Part II – handling Simulink/Stateflow
  - Simulink: type/clock inference and translation to Lustre
  - Stateflow: static checks and translation to Lustre

Agenda

- Part I – from synchronous models to implementations
  - Single-processor/single-task code generation
  - Multi-task code generation:
    - the Real-Time Workshop™ solution
    - a general solution
  - Implementation on a distributed platform:
    - General concerns
    - Implementation on a Kahn process network
    - Implementation on the Time Triggered Architecture
- Part II – handling Simulink/Stateflow
  - Simulink: type/clock inference and translation to Lustre
  - Stateflow: static checks and translation to Lustre

Code generation: single-processor, single-task

- Code that implements a state machine:

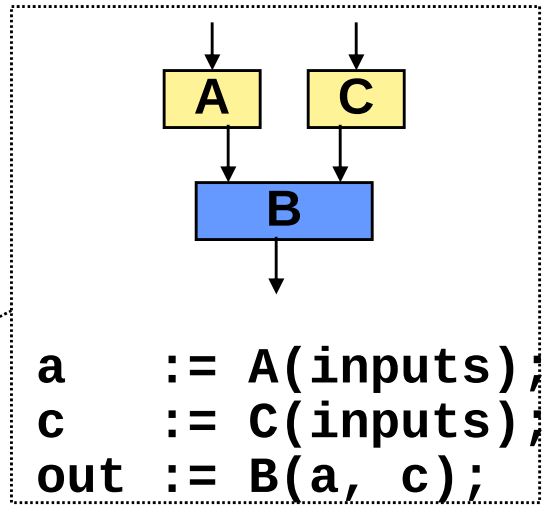



```
initialize;
repeat forever
  await trigger
  read inputs;
  compute next state
    and outputs;
  write outputs;
  update state;
end repeat;
```

Single-processor, single-tasking (1)

- One computer, no RTOS (or minimal), one process running
- Process has the following structure:

```
initialize state;
repeat forever
  await trigger;
  read inputs;
  compute new state and outputs;
  update state;
  write outputs;
end repeat;
```



- Trigger may be periodic or event-based
- Compute = “fire” all blocks in order (no cycles are allowed)
- Some major issues:
  - Estimate WCET (worst-case execution time)
    - “Hot” research topic, some companies also (e.g., AbsInt, Rapita, …)
  - Check that WCET <= trigger period (or minimum inter-arrival time)

Single-processor, single-tasking (2)

- One computer, no RTOS (or minimal), one process running
- Process has the following structure:

```
initialize state;
repeat forever
  await trigger;
  write (previous) outputs;   /* reduce jitter */
  read inputs;
  compute new state and outputs;
  update state;
end repeat;
```

- Other major issues:
  - Move from floating-point to fixed-point arithmetic
  - Evaluate the effect of jitter in outputs
  - Program size vs. memory (more/less “standard” compiler optimizations)
  - Handling causality cycles (dependencies within the same synchronous instant)
  - Modular code generation
  - …

To go further:

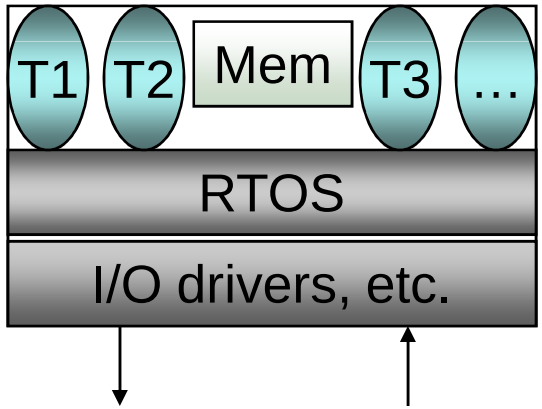
- Reinhard von Hanxleden's course slides:
  - http://www.informatik.uni-kiel.de/inf/von-Hanxleden/teaching/ws05-06/v-synch/skript.html#lecture14
- Pascal Raymond's course slides (in French):
  - http://www-verimag.imag.fr/~raymond/edu/compil-lustre.pdf
- "Compiling Esterel" book by Potop-Edwards-Berry (2007)

Agenda

- Part I – from synchronous models to implementations
  - Single-processor/single-task code generation
  - Multi-task code generation:
    - the Real-Time Workshop™ solution
    - a general solution
  - Implementation on a distributed platform:
    - General concerns
    - Implementation on a Kahn process network
    - Implementation on the Time Triggered Architecture
- Part II – handling Simulink/Stateflow
  - Simulink: type/clock inference and translation to Lustre
  - Stateflow: static checks and translation to Lustre

Code generation: single-processor, multi-task

- Multiple processes (tasks) running on the same computer
- Communicating by share memory (+some protocol)
- Real-time operating system (RTOS) handles scheduling



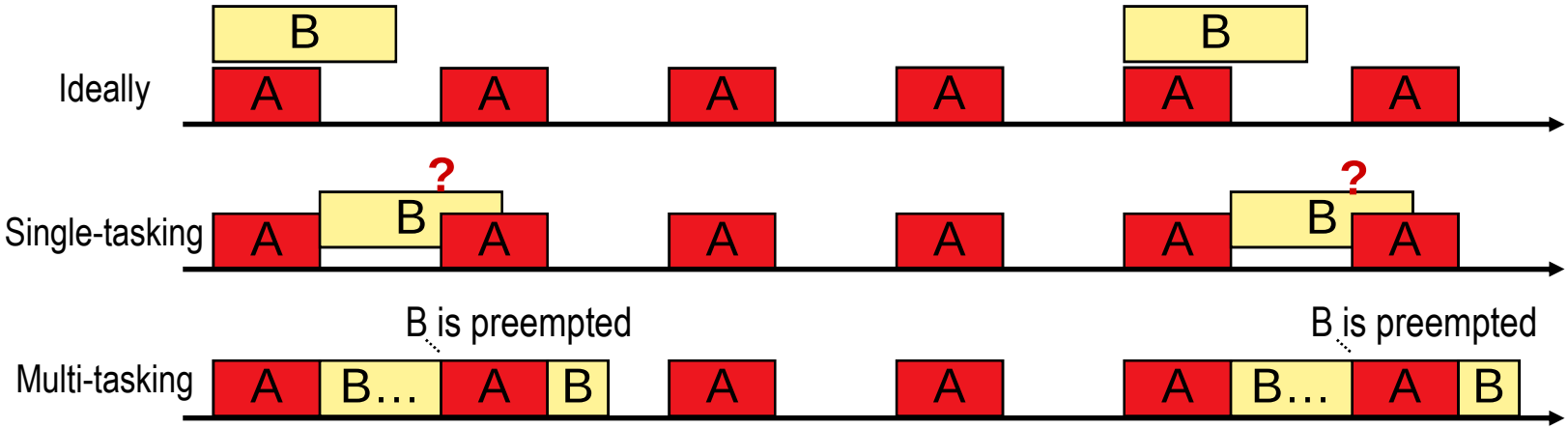
Question: why bother with multi-tasking? (since we could do single-task)

Code generation: single-processor, multi-task

- Multiple processes (tasks) running on the same computer
- Real-time operating system (RTOS) handles scheduling:
  - Usually fix-priority scheduling:
    - Each task has a fixed priority, higher-priority tasks preempt lower-priority tasks
  - Sometimes other scheduling policies
    - E.g., EDF = earliest deadline first
- Questions:
  - Why bother with single-processor, multi-tasking?
  - What are the challenges?

Single-processor, multi-tasking: why bother?

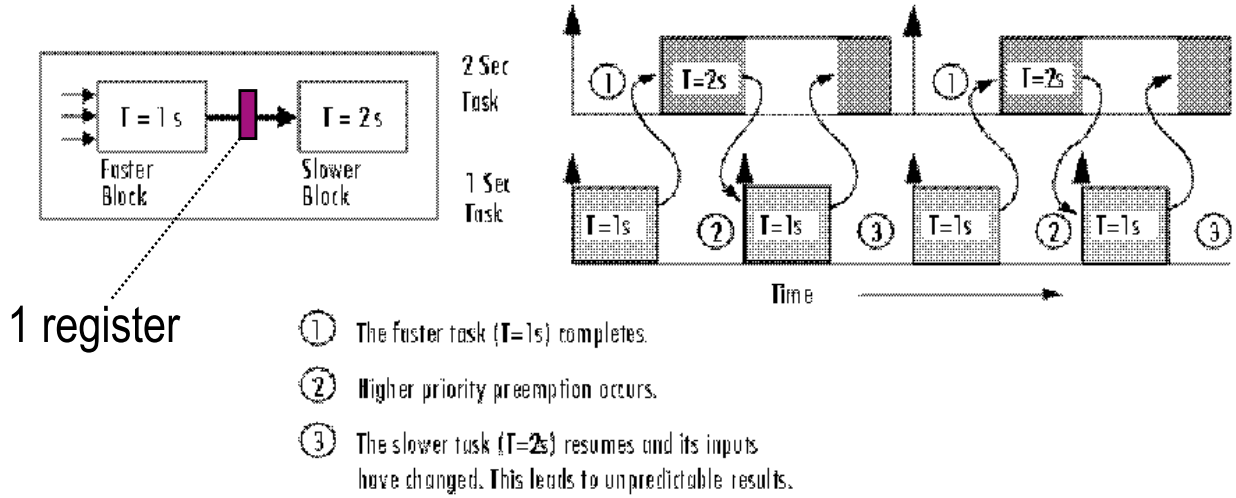
- Why bother?
  - For multi-rate applications: blocks running at different rates (triggers)
  - Example: block A runs at 10 ms, block B runs at 40 ms



WHAT IF TASKS COMMUNICATE?

Single-processor, multi-tasking issues

- Fast-to-slow transition (high-to-low priority) problems:

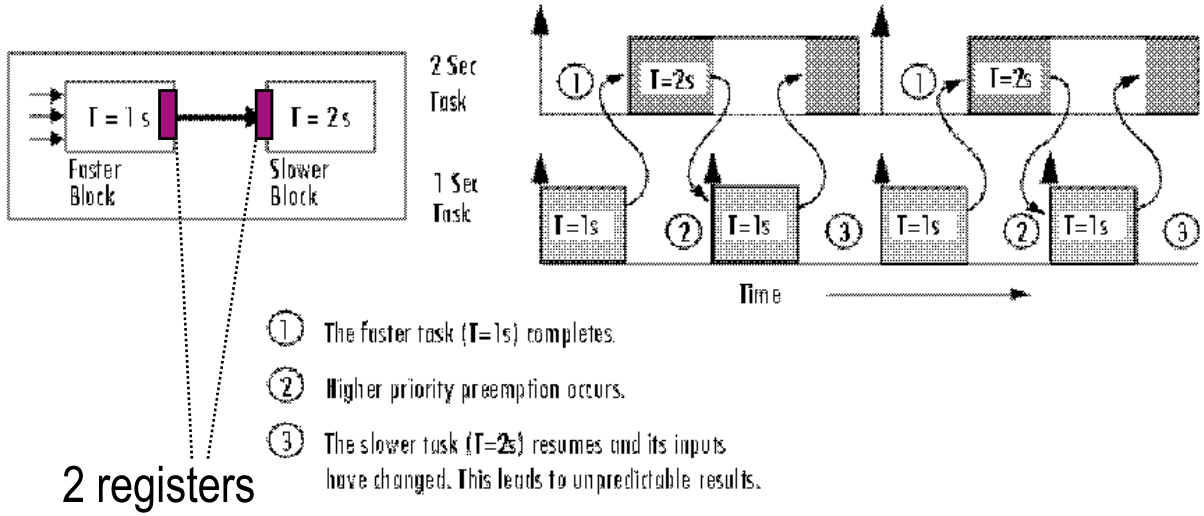


What would be the standard solution to this?

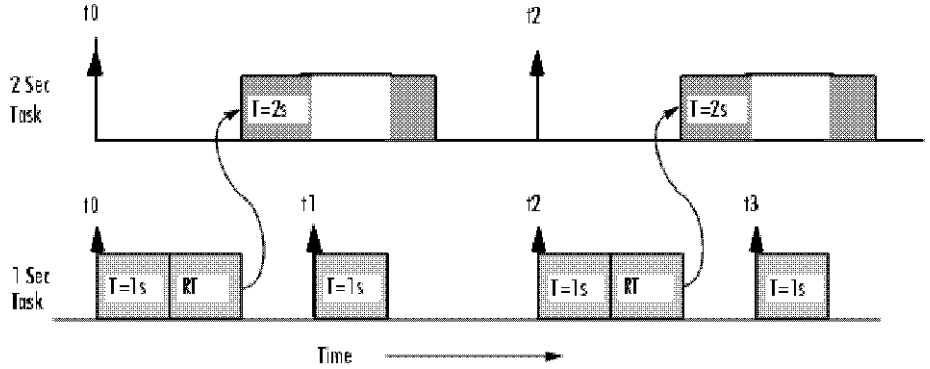
* Figures are cut-and-pasted from RTW User's Guide

Single-processor, multi-tasking issues

- Fast-to-slow transition (high-to-low priority) problems:



- RTW solution:
  - RT block
  - High priority
  - Low rate



Bottom-line: reader copies value locally when it starts

Does it work in general? Is it efficient?

- Not general:
  - Limited to periodic (in fact harmonic) arrival times
  - Fails for general (e.g., event-triggered) tasks
    - See examples later in this talk

- Not efficient:
  - Copying large data can take time…
  - What if there are many readers? Do they need to keep multiple local copies of the same data?

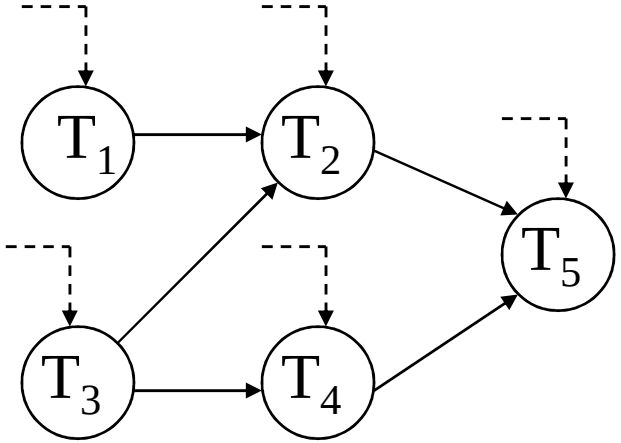
A better, general solution [ECRTS'04, EMSOFT'05,'06, TECS]

- The Dynamic Buffering Protocol (DBP)
  - Synchronous semantics preservation
  - General: applicable to any arrival pattern
    - Known or unknown
    - Time- or event- triggered
  - Memory optimal in all cases
  - Known worst case buffer requirements (for static allocation)

- Starting point: abstract synchronous model
  - Set of tasks
  - Independently triggered
  - Communicating
  - Synchronous ("zero-time") semantics

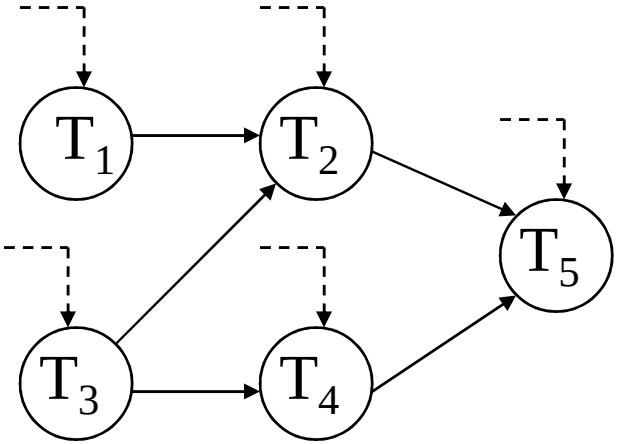
The model:
an abstraction of Simulink, Lustre, etc.

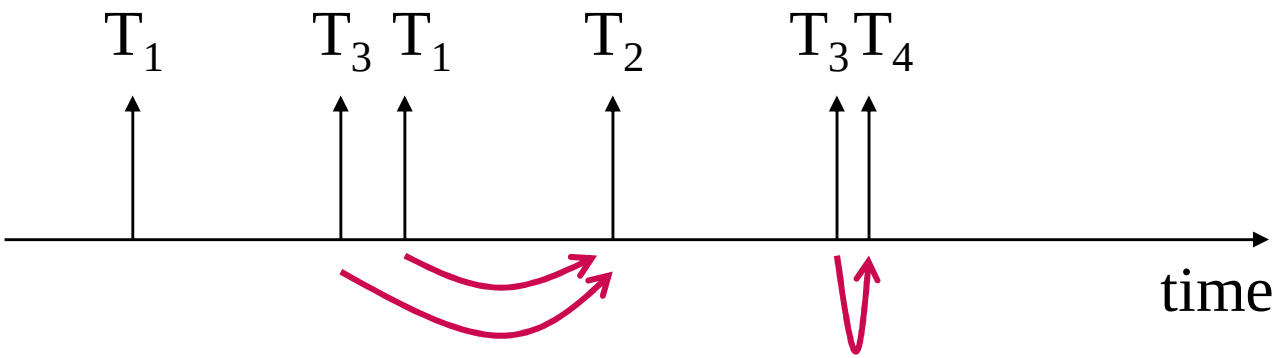
- A set of communicating tasks
- Time- or event-triggered



The model: semantics

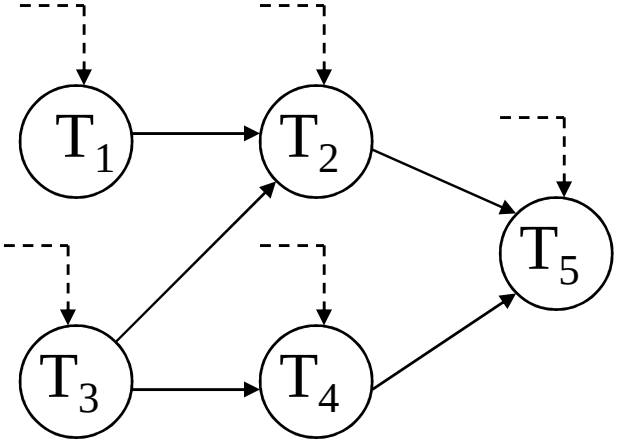
- Zero-time => “freshest” value

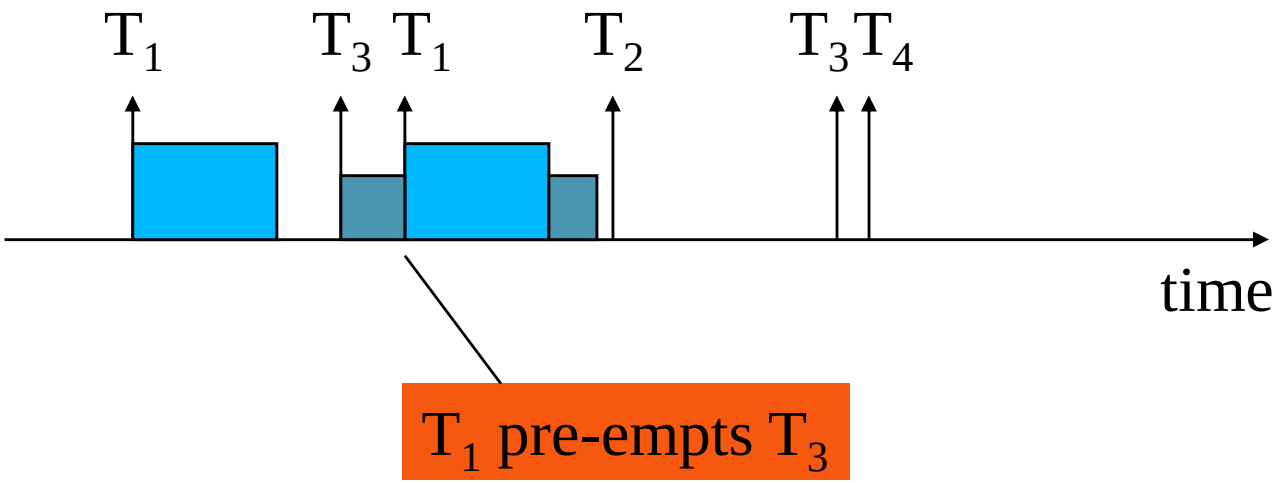




Execution on a real platform

- Execution takes time
- Pre-emption occurs





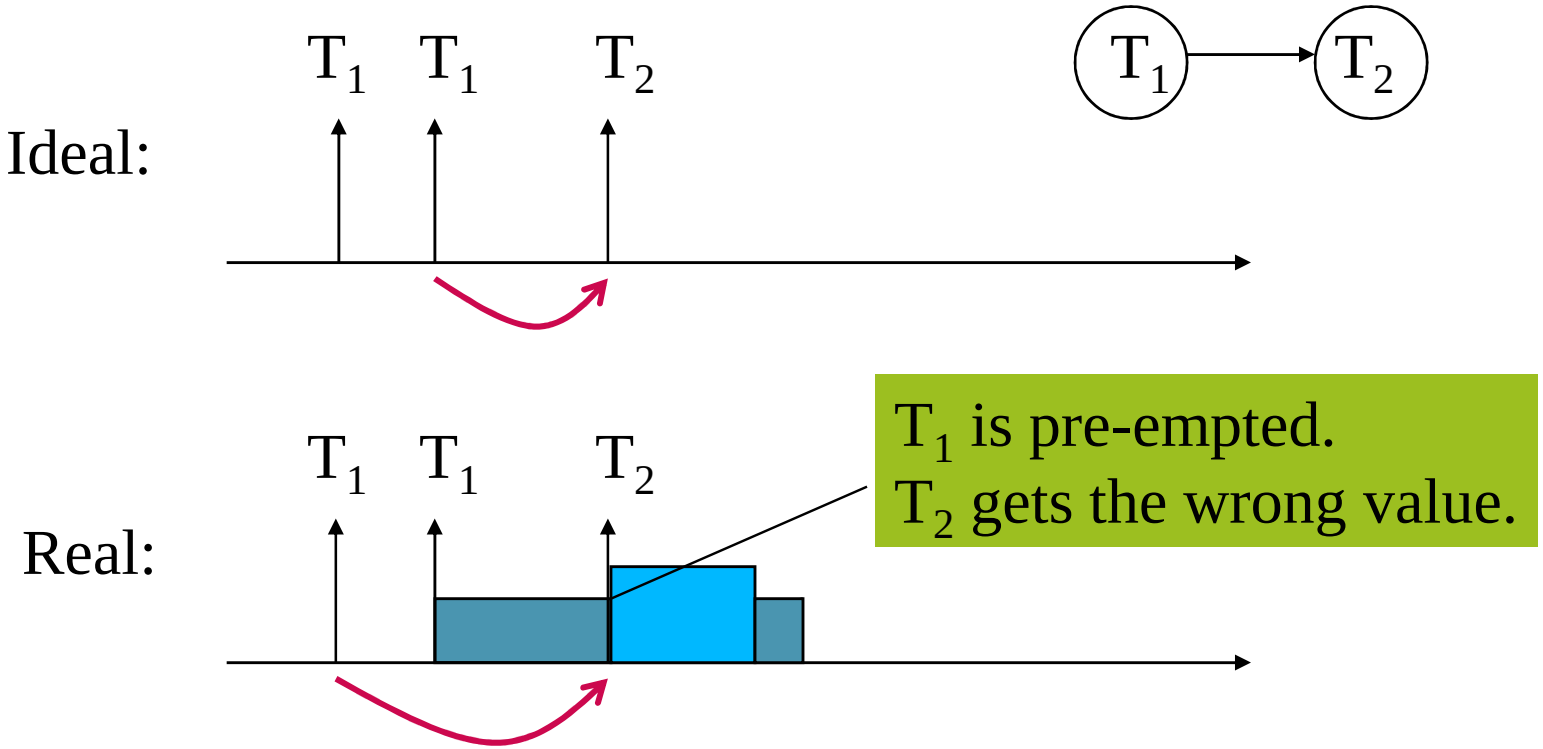
Assumption: schedulability

- *When a task arrives, all previous instances have finished execution.*

- How to check schedulability? Use scheduling theory!
- (will have to make assumptions on task arrivals)

Issues with a “naïve” implementation (1)

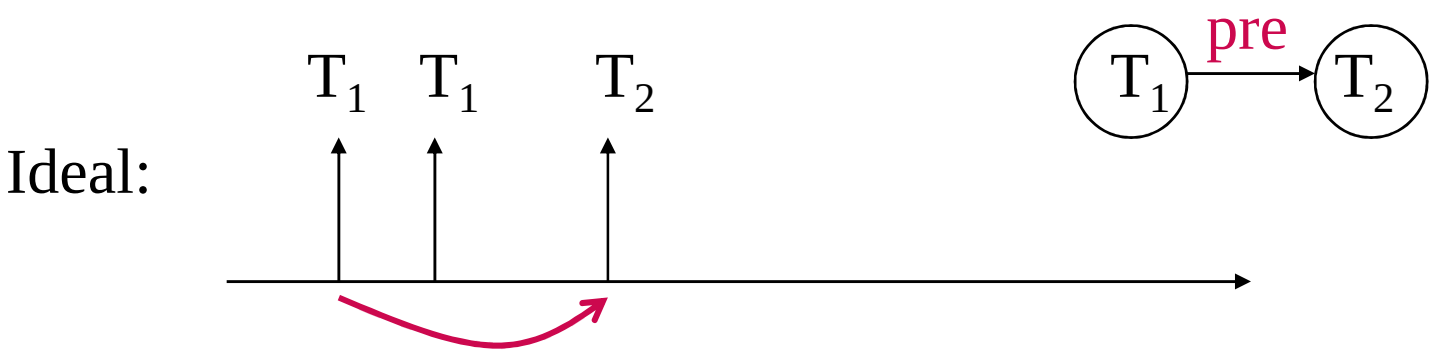
- Static-priority, $T_2 > T_1$



(*) “naïve” = atomic copy locally when task starts

Issues with a “naïve” implementation (1)

- Static-priority, $T_2 > T_1$

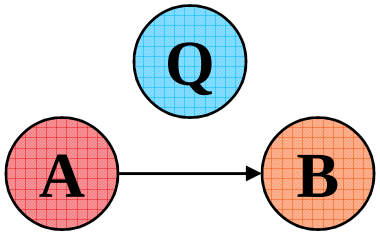


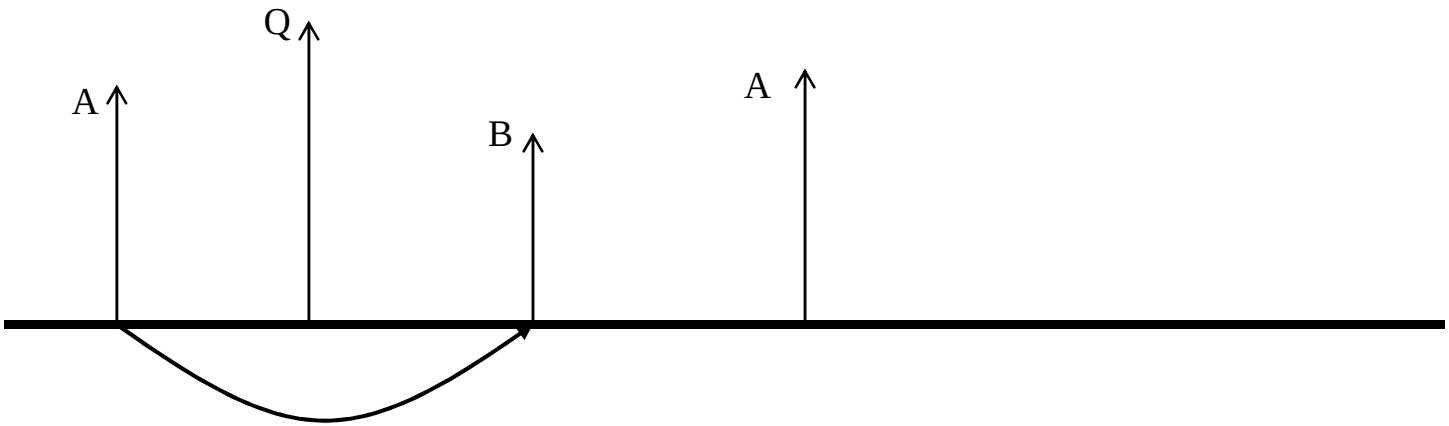
- Assumption: *if reader has higher priority than writer, then there is a unit-delay (“pre”) between them*.

- (RTW makes the same assumption)

Issues with a “naïve” implementation (2)

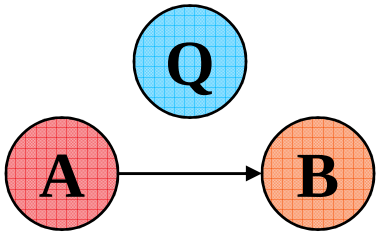
ideal semantics



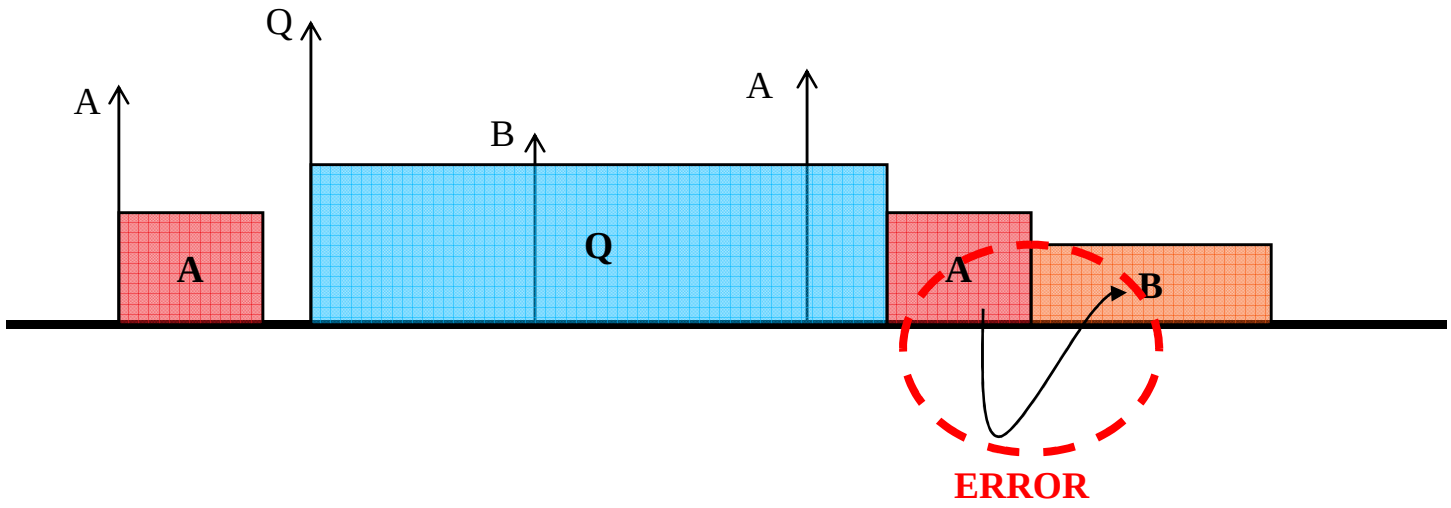


Issues with a “naïve” implementation (2)

real implementation



$Prio_Q > Prio_A > Prio_B$



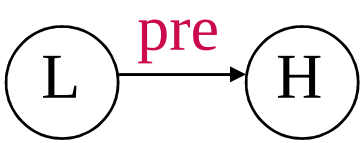
The DBP protocols

- Basic principle:
  - “Memorize” (implicitly) the **arrival order** of tasks

- Special case: one writer/one reader
- Generalizable to one writer/many readers (same data)
- Generalizable to general task graphs

One writer/one reader (1)

- Low-to-high case:



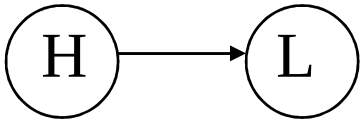
  - L keeps a double buffer `B[0,1 ]`
  - Two bits: `current, previous`
  - L writes to: `B[current ]`
  - H reads from: `B[previous ]`

  - When L arrives: `current := not current`
  - When H arrives: `previous := not current`

  - Initially: `current = 0, B[0 ]= B[1 ]= default`

One writer/one reader (2)

- High-to-low case:

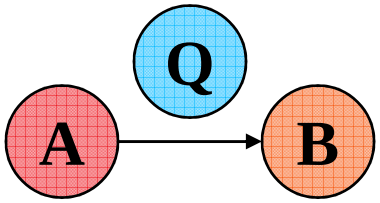


  - L keeps a double buffer `B[0,1 ]`
  - Two bits: `current, next`
  - H writes to: `B[next ]`
  - L reads from: `B[current ]`

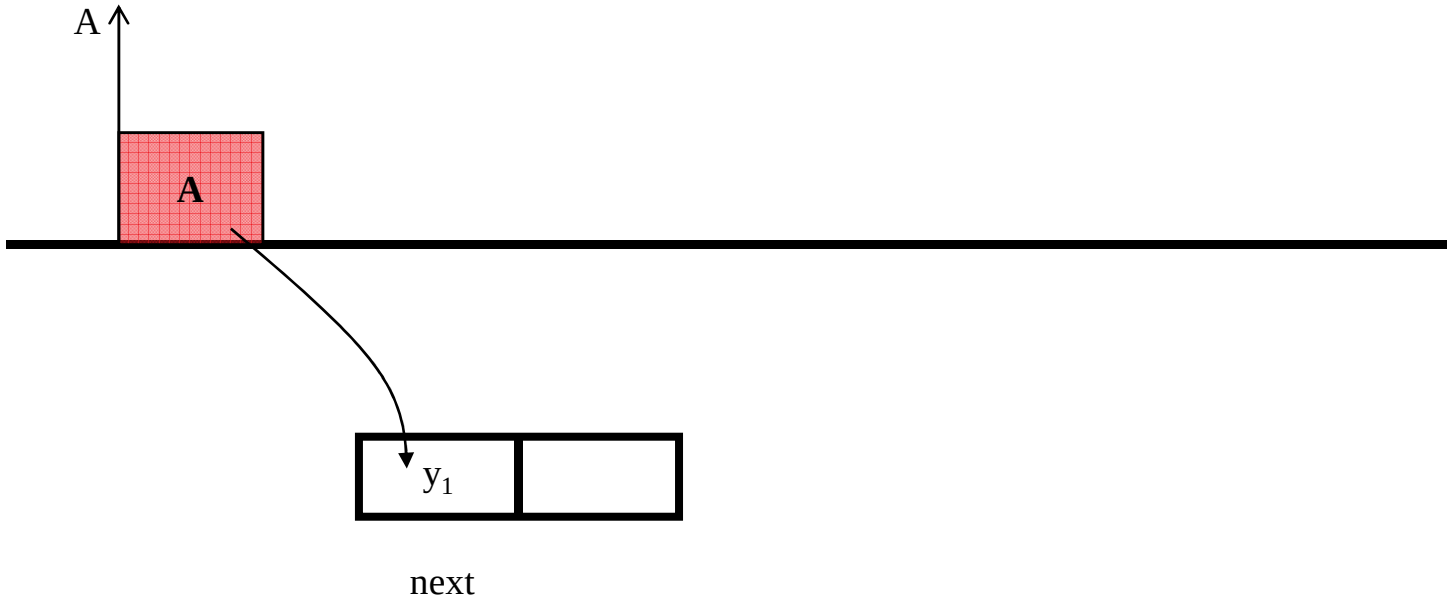
  - When L arrives: `current := next`
  - When H arrives:
```
if (current = next) then
        next := not next
```

  - Initially: `current=next=0, B[0 ]= B[1 ]= default`

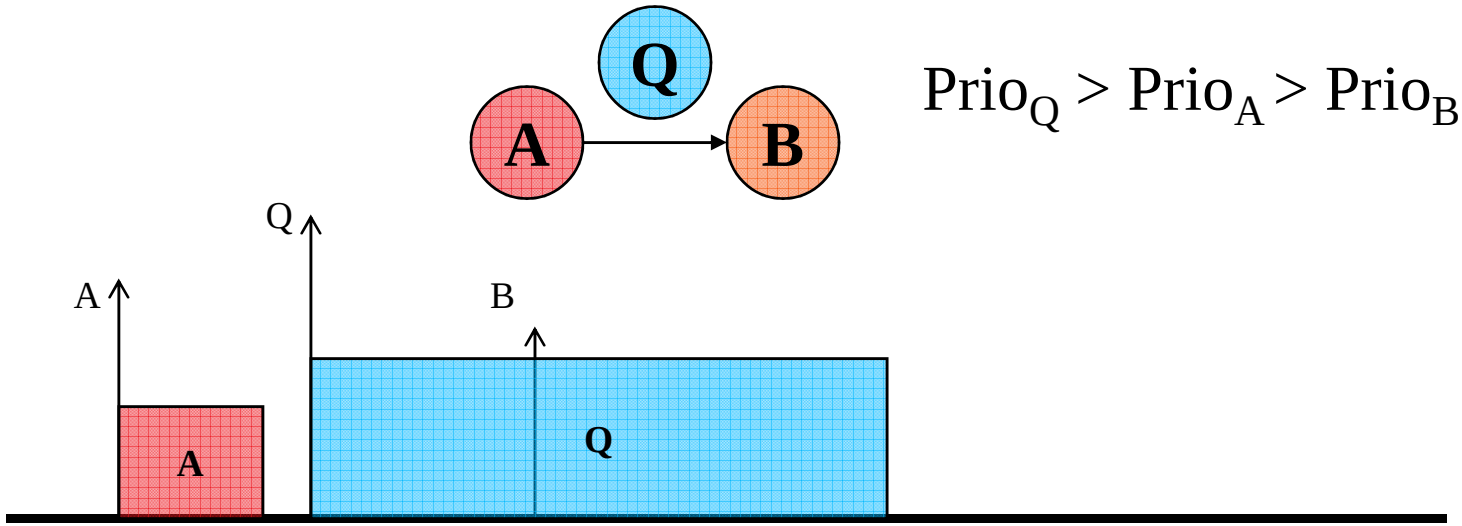
“hi2low” protocol demonstration



$Prio_Q > Prio_A > Prio_B$



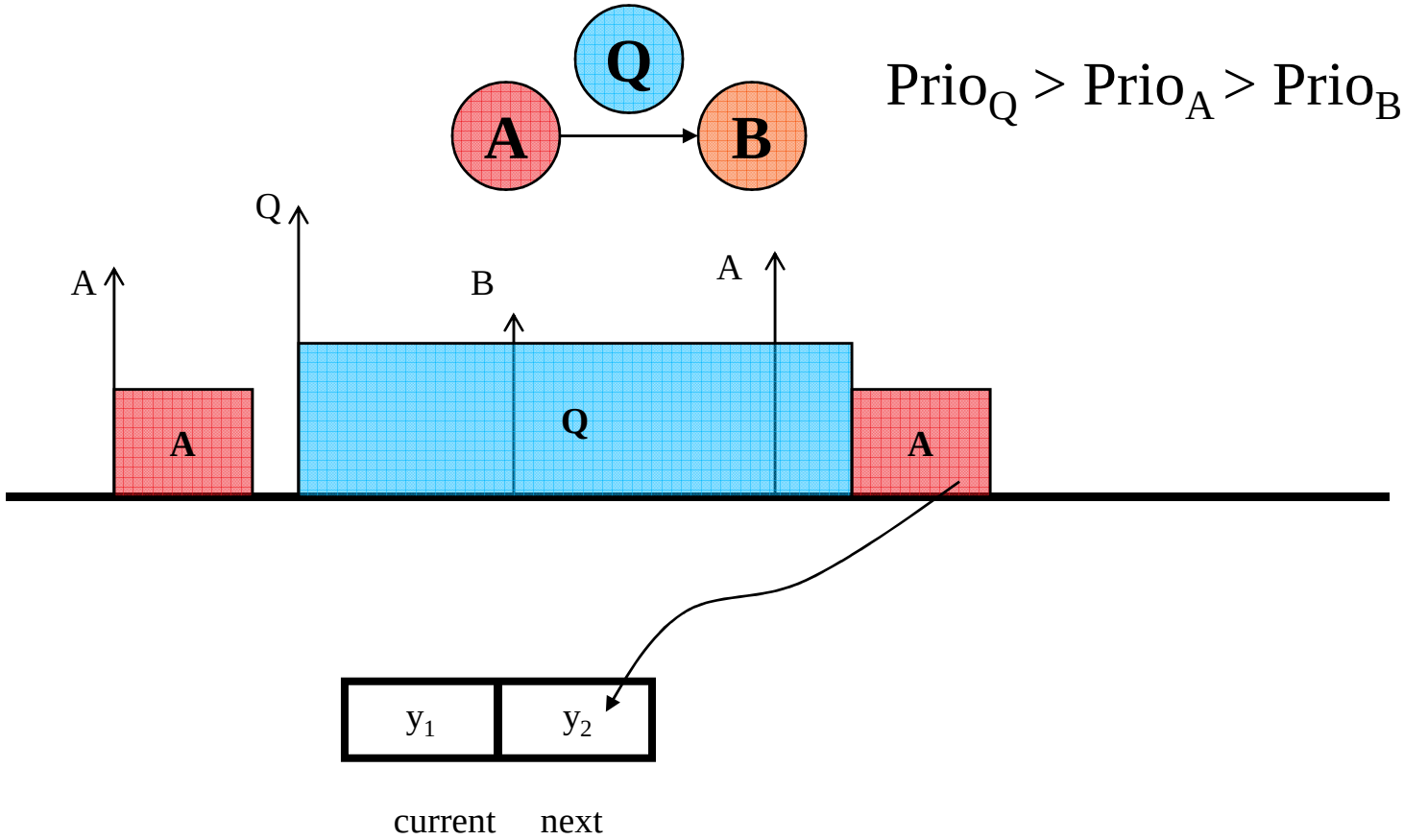
“hi2low” protocol demonstration



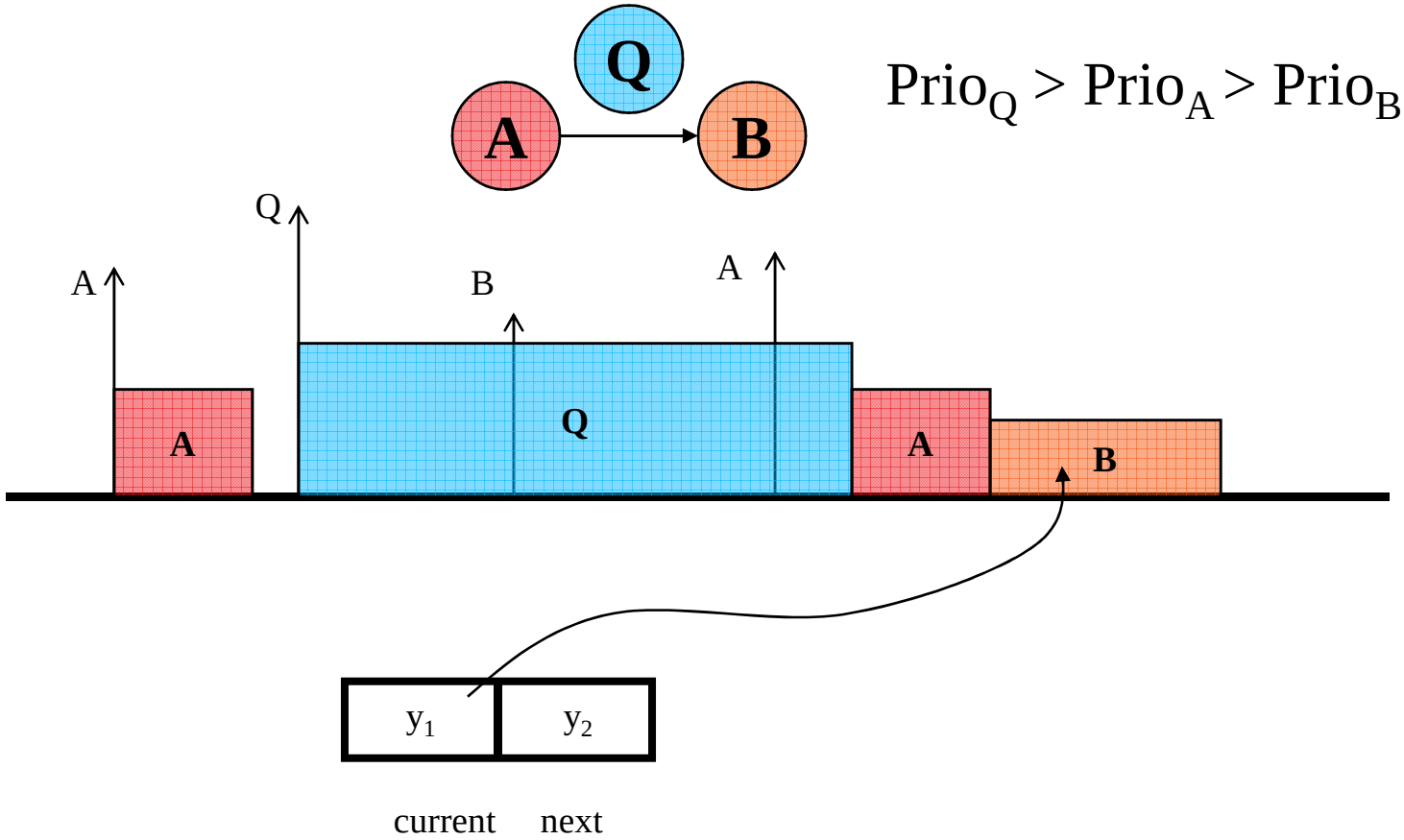
$Prio_Q > Prio_A > Prio_B$

next
current

“hi2low” protocol demonstration



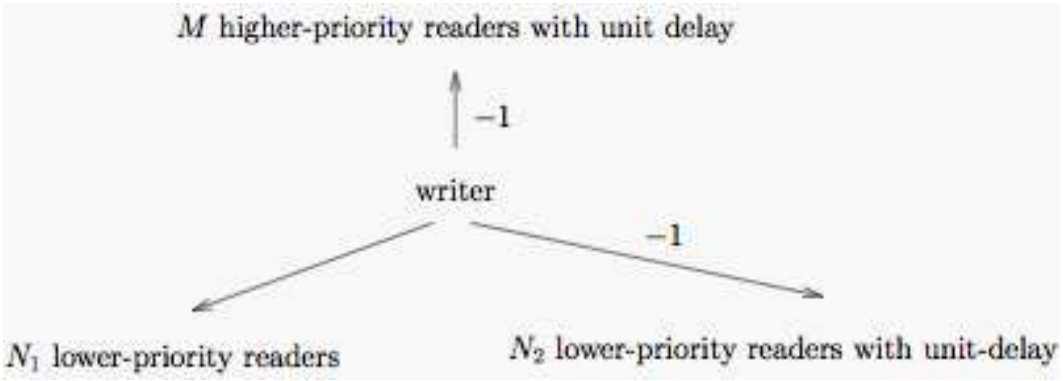
“hi2low” protocol demonstration



Dynamic Buffering Protocol (DBP)

- N_1 lower priority readers
- N_2 lower priority readers with unit-delay
- M higher priority readers (with unit-delay by default)

- *unit-delay* a delay to preserve the semantics
  - Read the previous input



The DBP protocol (1)

- Data structures:
  - Buffer array: `B[1..N+2] // stores the real data`
  - Pointer array: `H[1..M] // for higher-priority readers`
  - Pointer array: `L[1..N] // for lower-priority readers`
  - Two pointers: `current, previous`

- Writer
  - Release:

    `previous := current`

    `current := some j∈[1..N+2] such that B[j] is “free”`
  - Execution:

    write on `B[current]`

The DBP protocol (2)

- Lower-priority reader (without unit delay)
  - Release

    if unit-delay `L[i] := previous`

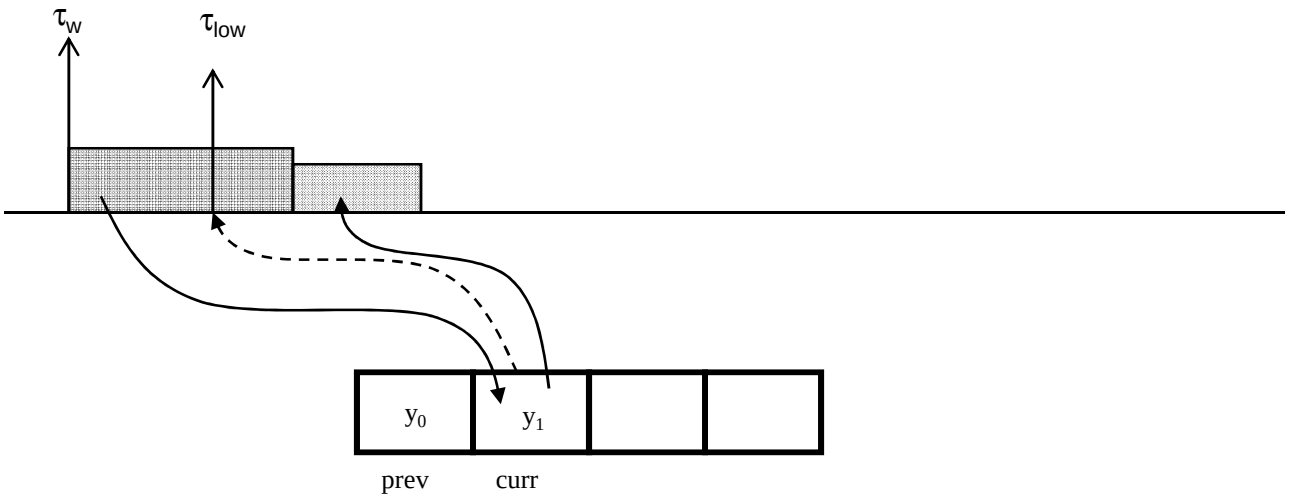
    else `L[i] := current`
  - Execution:

    read from `B[L[i]]`
- Higher-priority reader (with unit delay)
  - Release

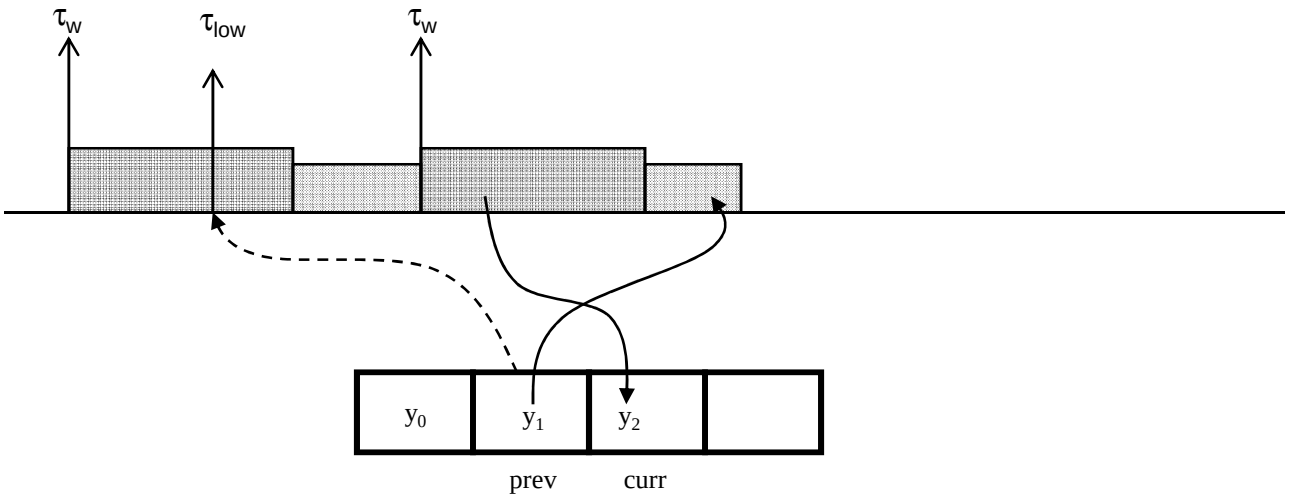
    `H[i] := previous`
  - Execution

    read from `B[H[i]]`

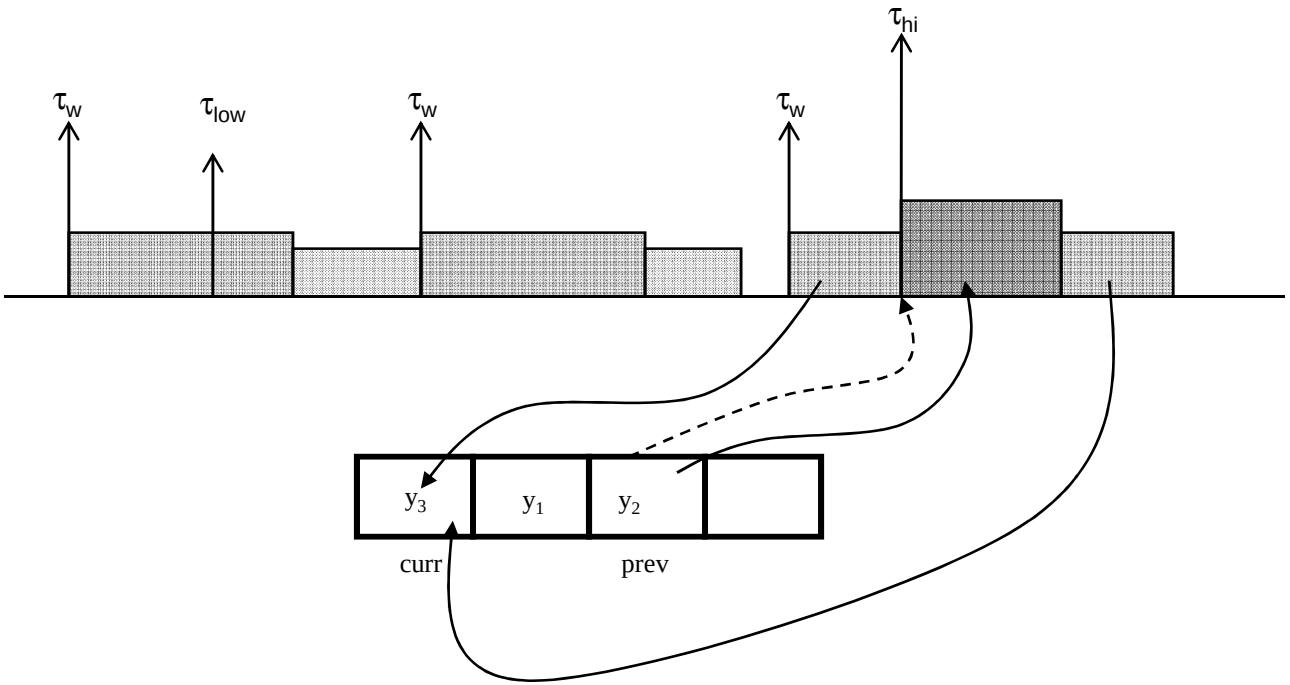
Example of usage of DBP



Example of usage of DBP



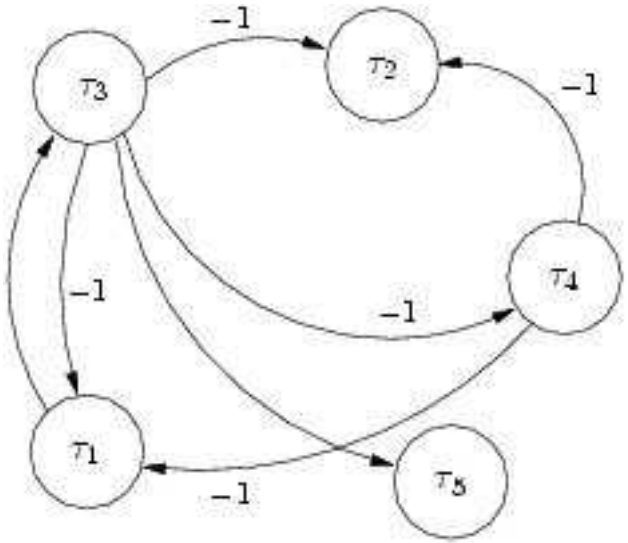
Example of usage of DBP



Savings in memory

- One writer → one reader : 14 buffers

- DBP
  - τ_1 2 buffers
  - τ_3 4 buffers
  - τ_4 2 buffers
- Total: 8 buffers



Worst case buffer consumption

- DBP never uses more than N_1+N_2+2 buffers
  - N_1 lower priority readers
  - N_2 lower priority readers with a *unit-delay*
  - M higher priority readers (only contribute at most 1 buffer)

Optimality

- DBP is memory optimal in **any** arrival execution
- Let ρ be some execution
  - `Maybeneeded(ρ,t)`
    - Used now
    - May be used until next execution of the writer
  - `DBP_used(ρ,t)`
    - buffers used by the DBP protocol

- Theorem: for all ρ, t

  `DBP_used(ρ,t)` $\subseteq$ `maybeneeded(ρ,t)`

Optimality for known arrival pattern

- DBP is ***non-clairvoyant***
  - Does not know future arrivals of tasks
  - => it may keep info for a reader that will not arrive until the next execution of the writer: redundant
- How to make DBP optimal when task arrivals are known?
  - E.g.: multi-periodic tasks
- Two solutions:
  - Dynamic: for every writer, store output only if it will be needed (known since, readers' arrivals are known)
  - Static: Simulate arrivals tasks until *hyper-period* (if possible)
- Standard time vs. memory trade-off

Conclusions and perspectives (part I)

- Dynamic Buffering Protocol
  - Synchronous semantics preservation
  - Applicable to any arrival pattern
    - Known or unknown
    - Time or event triggered
  - Memory optimal in all cases
  - Known worst case buffer requirements (for static allocation)

- Relax schedulability assumption
- More platforms (in the model based approach)
  - CAN, Flexray, …
- Implement the protocols and experiment

- BIG QUESTION: how much does all this matter for control???

Agenda

- Part I – from synchronous models to implementations
  - Single-processor/single-task code generation
  - Multi-task code generation:
    - the Real-Time Workshop™ solution
    - a general solution
  - Implementation on a distributed platform:
    - General concerns
    - Implementation on a Kahn process network
    - Implementation on the Time Triggered Architecture
- Part II – handling Simulink/Stateflow
  - Simulink: type/clock inference and translation to Lustre
  - Stateflow: static checks and translation to Lustre

General concerns

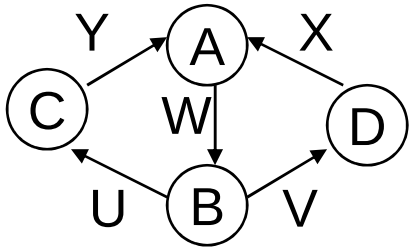
- What semantics to preserve?
  - Sequence of values? Synchronism? Both? None?
- How to achieve real-time constraints?
- How to distribute computation on the execution nodes?
- How to distribute communication?
- How to solve computation/communication trade-offs?
  - E.g., duplicate computation to avoid communication
- How to achieve fault-tolerance?
- And many, many more:
  - Local and end-to-end scheduling, SW architecture, buffer sizing, …

Agenda

- Part I – from synchronous models to implementations
  - Single-processor/single-task code generation
  - Multi-task code generation:
    - the Real-Time Workshop™ solution
    - a general solution
  - Implementation on a distributed platform:
    - General concerns
    - Implementation on a Kahn process network
    - Implementation on the Time Triggered Architecture
- Part II – handling Simulink/Stateflow
  - Simulink: type/clock inference and translation to Lustre
  - Stateflow: static checks and translation to Lustre

Kahn Process Networks [G. Kahn, “The semantics of a simple language for parallel programming”, 1974]

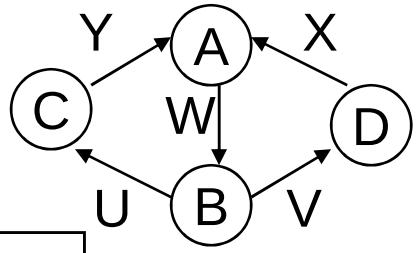
- A network of processes:
  - A, B, C, D are processes
  - X, Y, U, V, W are channels of the network



- What is a network?
  - Point-to-point channels between processes
  - Each channel is a lossless, FIFO queue of unbounded length
  - No other means of communication between processes

- What is a process?
  - A sequential program (could be written in C, C++, etc.)
  - Uses “wait” (blocking read) and “send” (non-blocking write) primitives to receive/send data from/to its input/output channels

Example of process

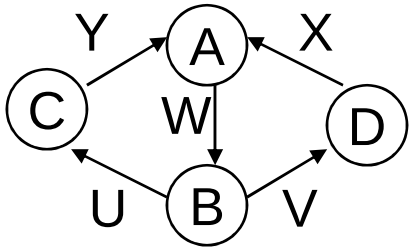



```
Process A(integer in X, Y; integer out W)
  Begin
       integer i;
       boolean b := true;
       while (true) do
         i := if b then wait(X) else wait(Y);
         send i on W;
         b := not b;
       end while;
  End.
```

Main results of Kahn

- The behavior of a KPN is deterministic:
  - It does not depend on the execution order of processes (modeling execution speed, transmission delays, …)
  - Behavior = sequences of input/output values of each process
- How to prove it:
  - View each channel as carrying a (finite or infinite) sequence of values
  - Order sequences by prefix-order
  - Set of sequences is then a CPO (bottom is the empty sequence)
  - Then:
    - Kahn processes are continuous functions in this CPO
    - Network is a set of fix-point equations on these functions
    - (From continuity) the set of equations has a (unique) least fixpoint
    - This least fixpoint is the semantics

Example of fixpoint equations




```
W     = A(X,Y)
(U,V) = B(W)
Y     = C(U)
X     = D(V)
```

Questions – take as homework!

- Kahn processes and continuous functions
  - Why are Kahn processes continuous functions?
  - What processes would not be continuous?
  - E.g., suppose we had a new primitive: wait-either(X,Y) that blocks until a value is received on EITHER of X, Y. Would processes still be continuous? Can you think of other primitives that could make processes non-continuous?
  - Are there “good” (continuous, other) functions not expressed as Kahn processes?
- How to implement synchronous programs on KPN?
  - E.g., take Lustre programs
  - Suppose the program is a “flat” network of nodes
  - Suppose each Lustre node is to be mapped into a separate Kahn process
  - What next?
  - What semantics does your implementation method preserve?

Agenda

- Part I – from synchronous models to implementations
  - Single-processor/single-task code generation
  - Multi-task code generation:
    - the Real-Time Workshop™ solution
    - a general solution
  - Implementation on a distributed platform:
    - General concerns
    - Implementation on a Kahn process network
    - Implementation on the Time Triggered Architecture
- Part II – handling Simulink/Stateflow
  - Simulink: type/clock inference and translation to Lustre
  - Stateflow: static checks and translation to Lustre

TTA: the Time Triggered Architecture [Kopetz et al]

- A distributed, synchronous, fault-tolerant architecture
  - Distributed: set of processor nodes + bus
  - Time-triggered:
    - static TDMA bus access policy
    - clock synchronization
  - Fault-tolerant: membership protocol built-in
  - Precursor of **FlexRay**

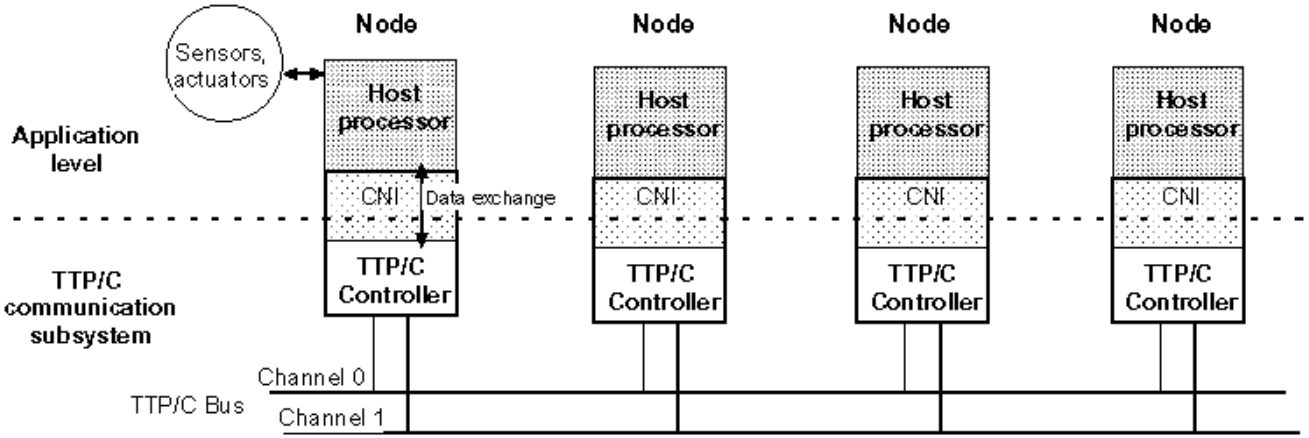


Figure 2.1: TTA Cluster

From Lustre to TTA

- The good news:
  - TTA is synchronous
  - No problems of clock synchronization
  - Synchronous semantics of Lustre can be preserved

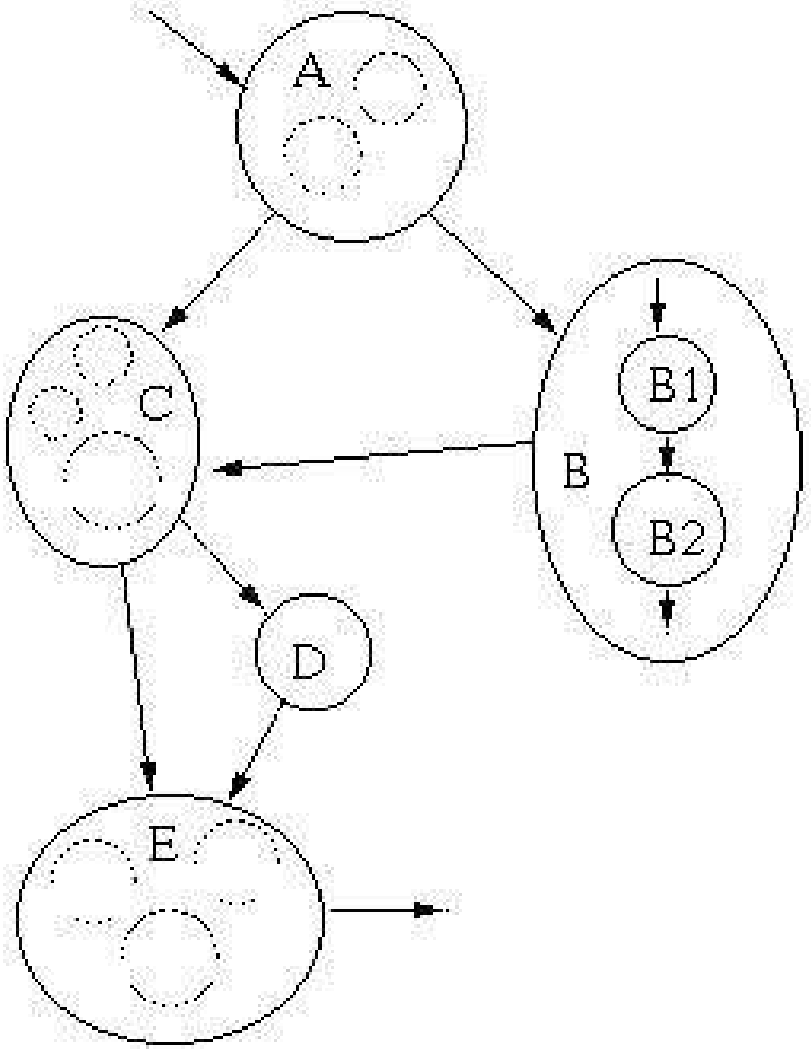
- The bad news: non-trivial resource-allocation problems
  - Decomposition of Lustre program into tasks
  - Mapping of tasks to processors
  - Scheduling of tasks and messages
  - Code (and glue code) generation

- Auxiliary (difficult) problem:
  - WCET analysis

- To “help” the compiler: Lustre extensions (“pragmas”) [LCTES’03]
  - Real-time primitives (WCET, deadlines, …)
  - Distribution primitives (user-defined mapping)

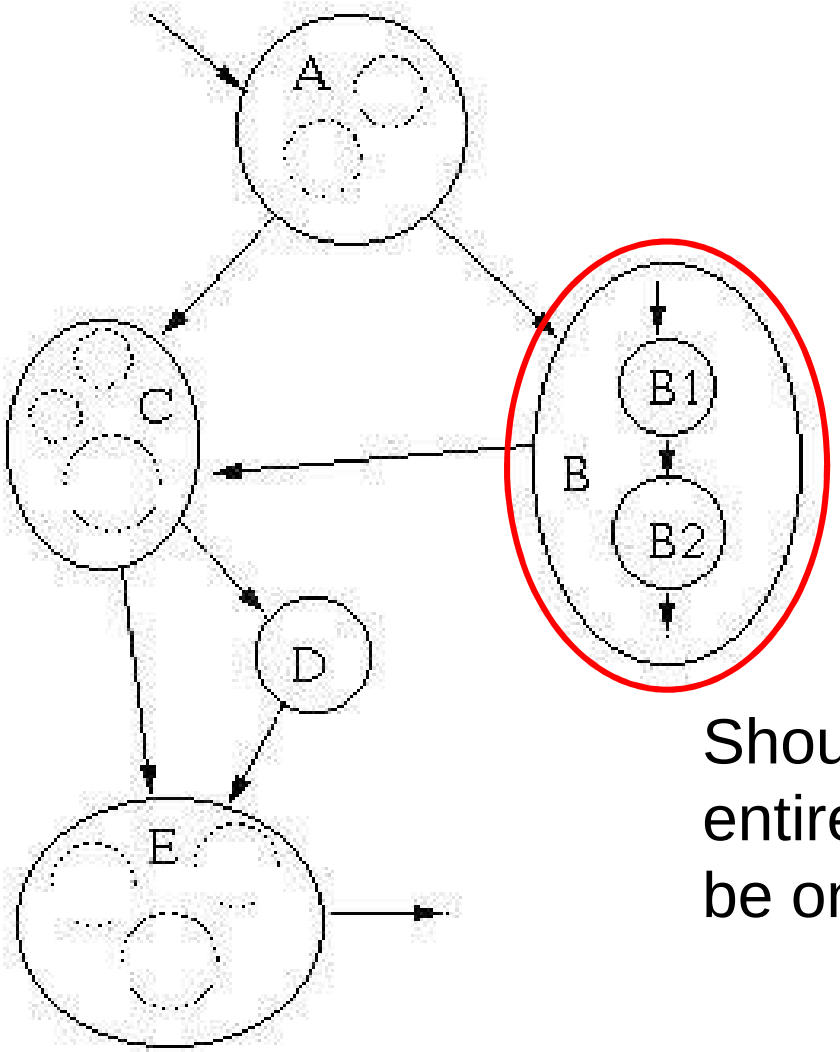
Decomposition

Lustre program:



Decomposition

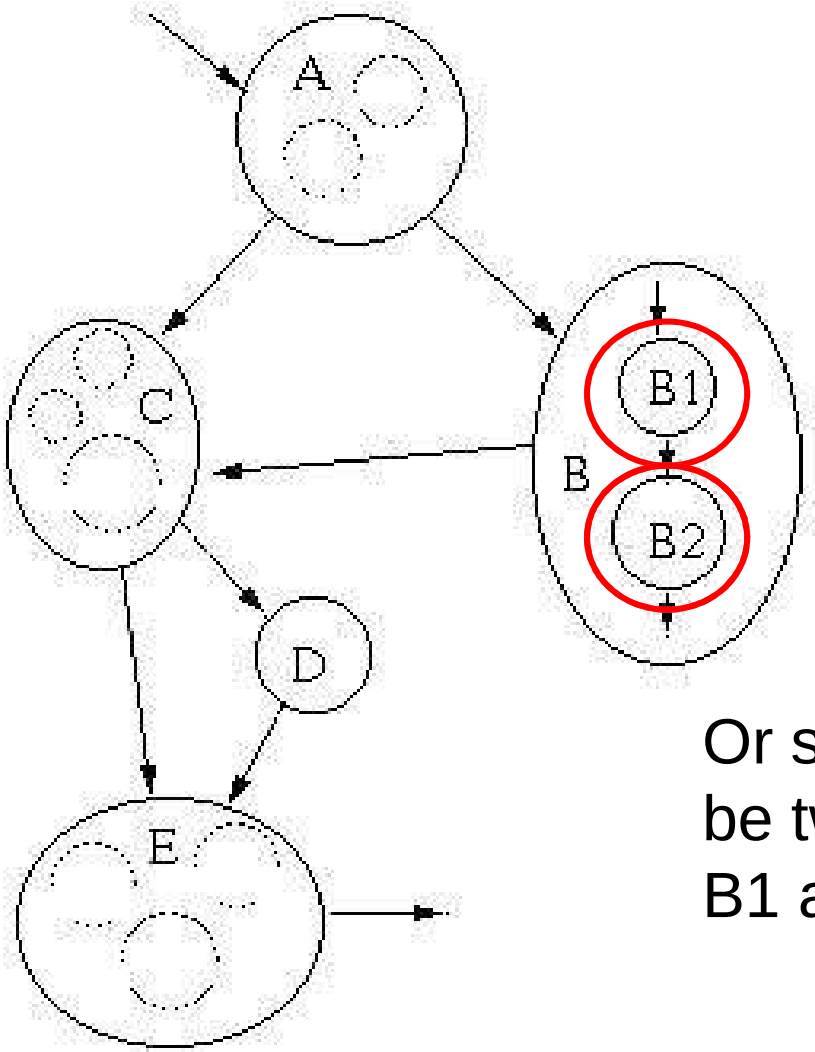
Lustre program:



Should the entire node B be one task?

Decomposition

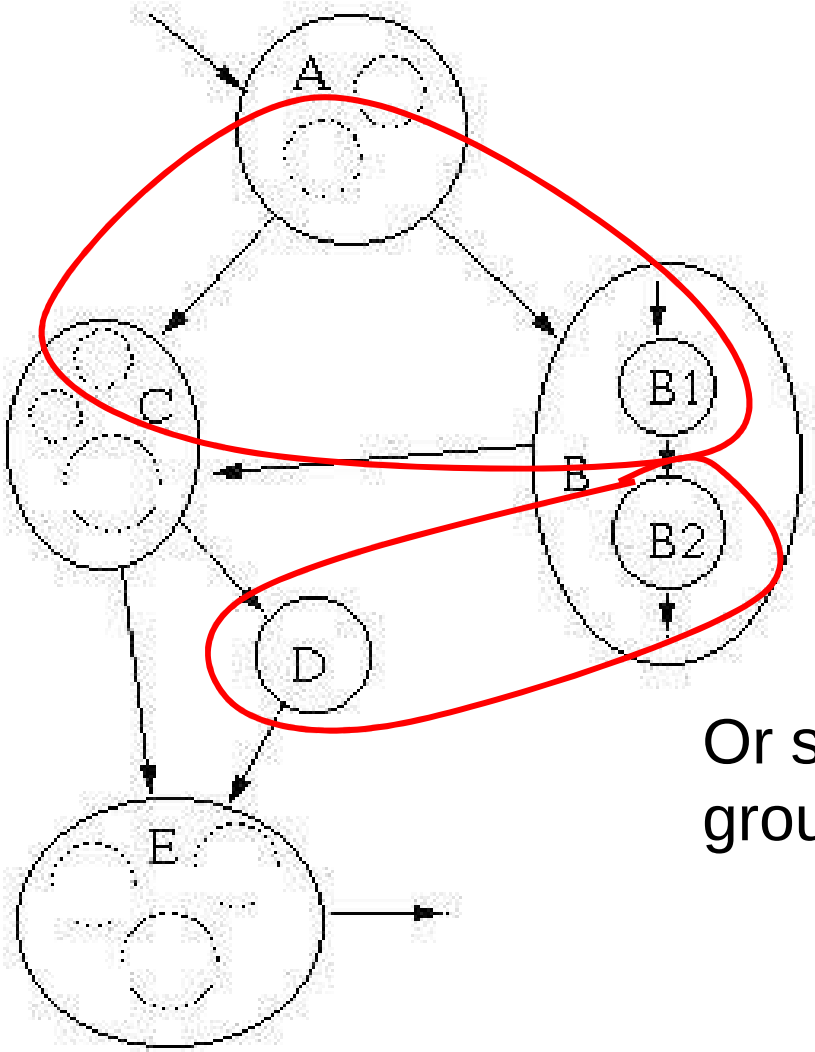
Lustre program:



Or should there be two tasks B1 and B2 ?

Decomposition

Lustre program:

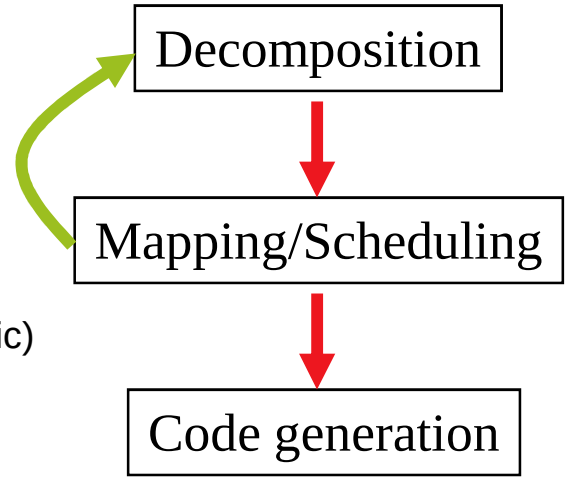


Or some other grouping ?

Decomposition

- Two extremes:
  - One task per processor: cheap but too coarse
    - perhaps no feasible schedule (pre-emption not allowed).
  - One task for every Lustre operator: fine but too costly
    - too many tasks, combinatorial explosion.

- Our approach:
  - Start with coarse partition.
  - Refine when necessary: feedback.
  - Feedback: heuristics
    - Split task with largest WCET
    - Split task that blocks many others
    - ...
    - (unpublished, in PhD thesis of Adrian Curic)



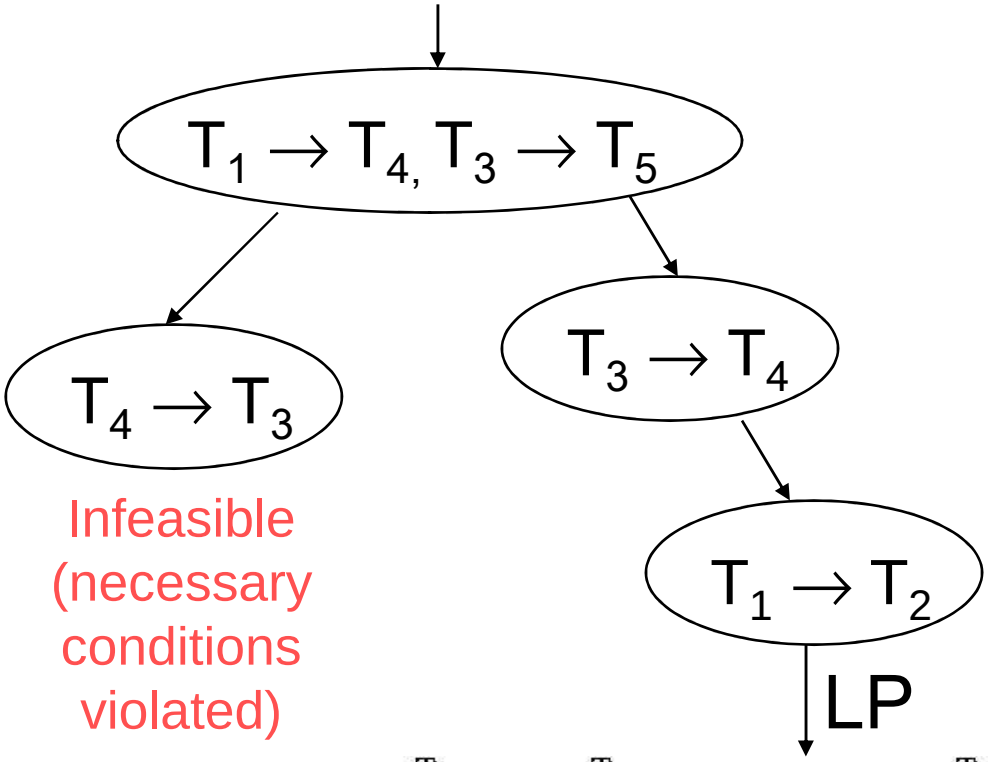
Scheduling

- Schedule tasks on each processor.
- Schedule messages on the bus.

- Static TDMA schedules (both for bus and processors).
- No pre-emption.
- TTA-specific constraints.
- Problem NP-hard.

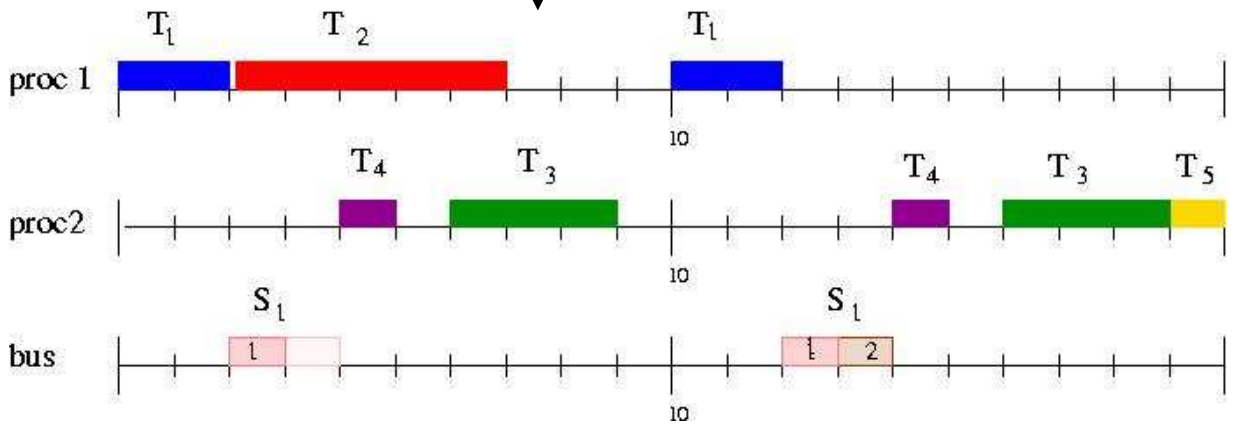
- Algorithm:
  - Branch-and-bound to fix order of tasks/messages.
  - Solve a linear program on leaves to find start times.
  - Ensures deadlines are met ∀ possible execution time.

Scheduling algorithm

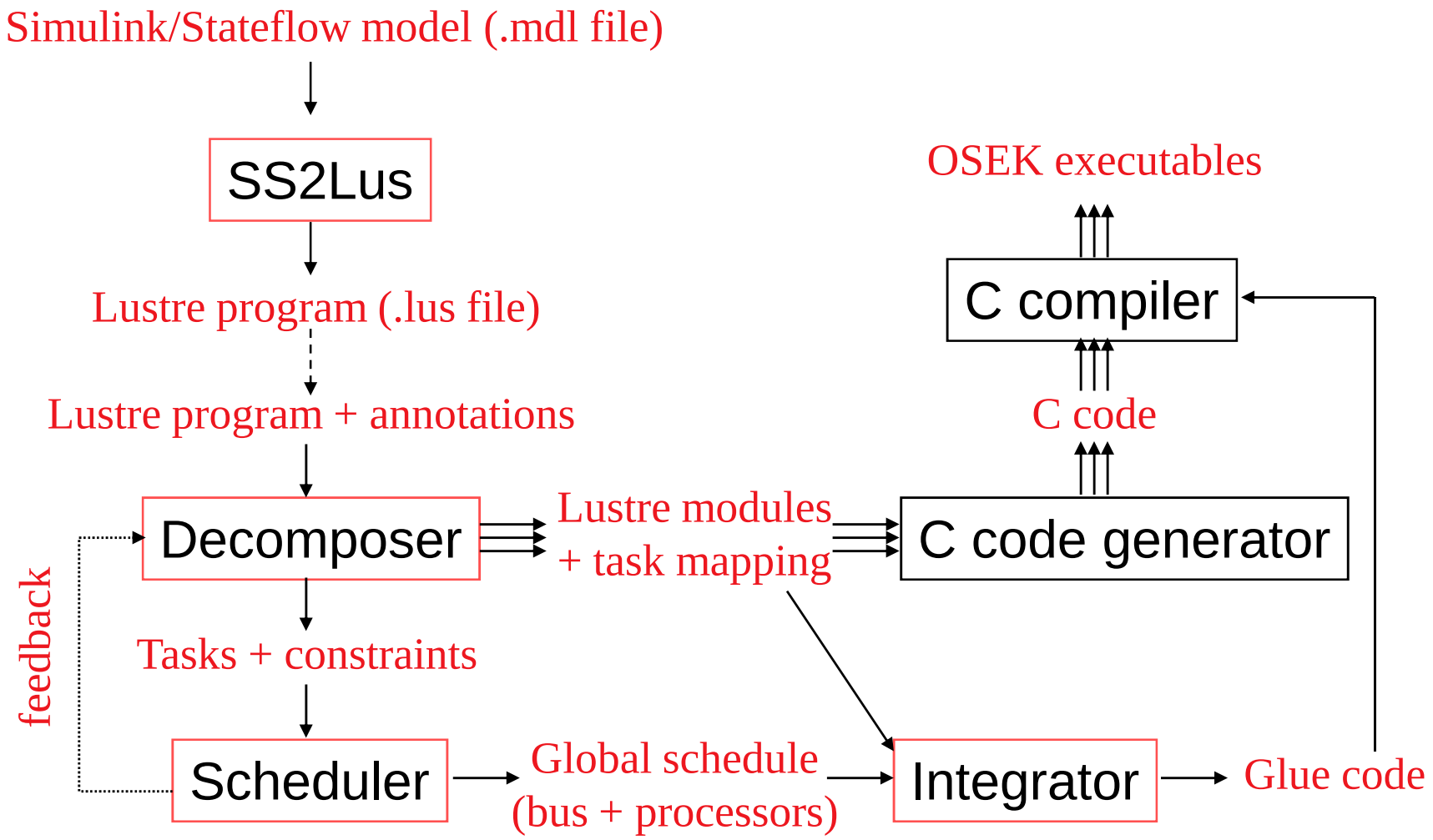


Infeasible (necessary conditions violated)

total order



Tool chain


Simulink/Stateflow model (.mdl file)
SS2Lus
Lustre program (.lus file)
Lustre program + annotations
Decomposer
Tasks + constraints
Scheduler
feedback
Lustre modules
+ task mapping
Global schedule
(bus + processors)
C code generator
C code
C compiler
OSEK executables
Integrator
Glue code
: currently manual
: on-going work

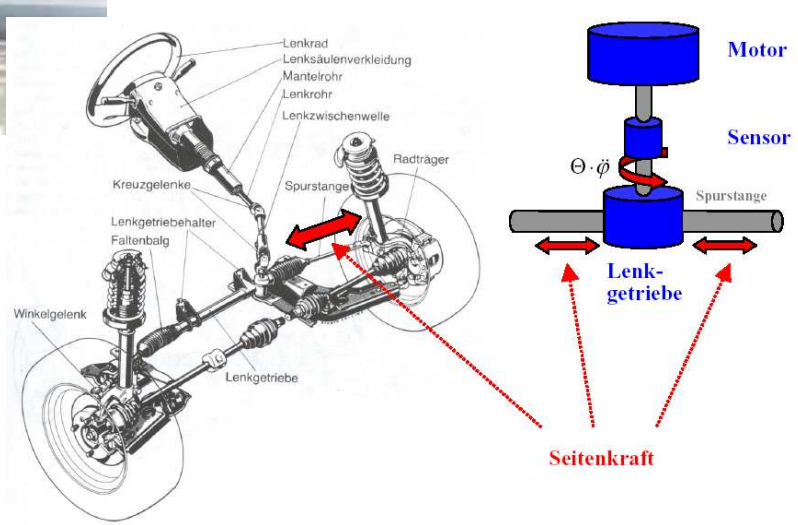
Case studies

- Two case studies from Audi.
  - A warning-filtering system:
    - 6 levels, 20 subsystems, 113 total blocks.
    - 800 lines of generated Lustre code.
  - An autonomous steer-by-wire application:
    - 6 levels, 18 subsystems, 157 total blocks.
    - 387 lines of generated Lustre code.
    - Demo-ed in final NEXT TTA review (Jan ‘04).

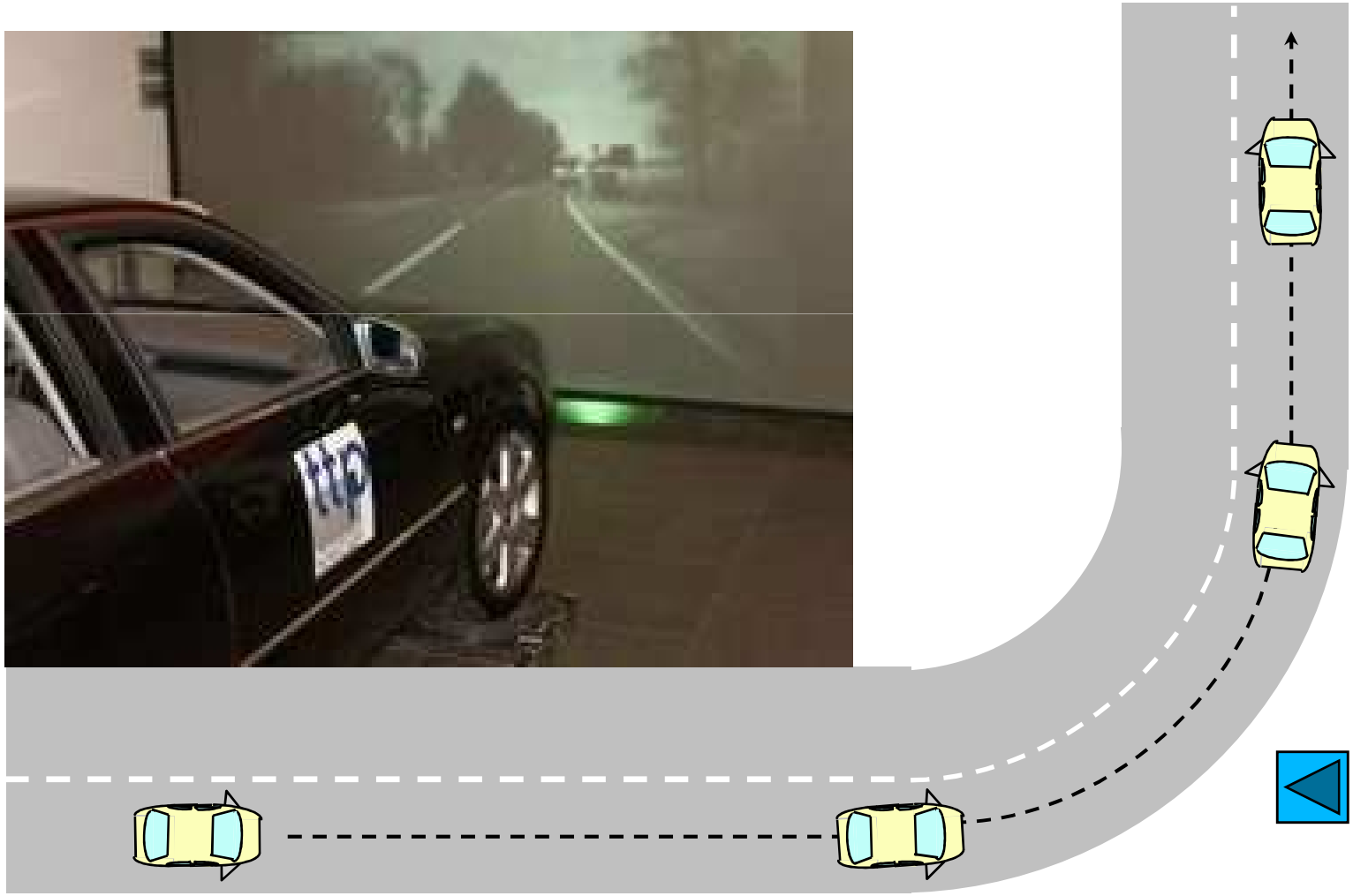
Autonomous steer-by-wire

Equipment:

- cameras/imaging
- steering actuator
- TTA network
- MPC555 nodes



Autonomous steer-by-wire

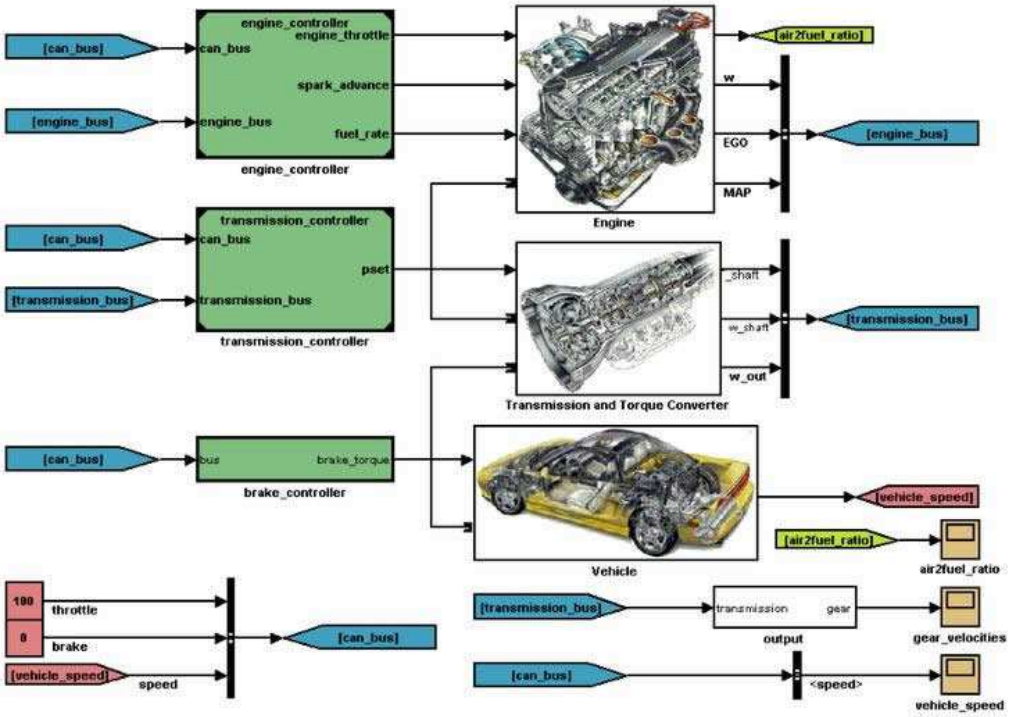


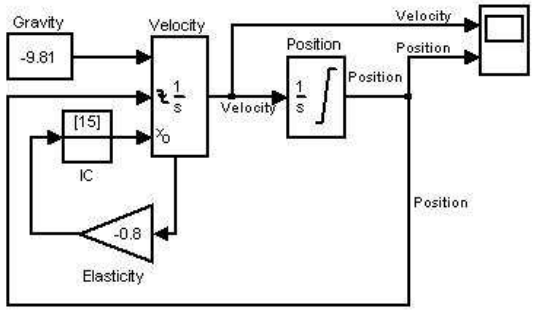
Agenda

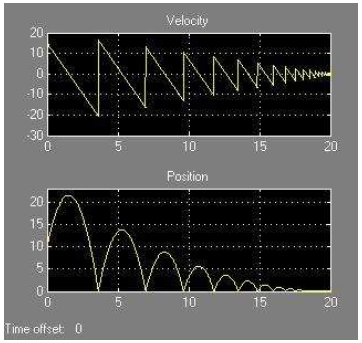
- Part I – from synchronous models to implementations
  - Single-processor/single-task code generation
  - Multi-task code generation:
    - the Real-Time Workshop™ solution
    - a general solution
  - Implementation on a distributed platform:
    - General concerns
    - Implementation on a Kahn process network
    - Implementation on the Time Triggered Architecture

- Part II – handling Simulink/Stateflow
  - Simulink: type/clock inference and translation to Lustre
  - Stateflow: static checks and translation to Lustre

Simulink™


[can_bus]
engine_controller
engine_throttle
can_bus
spark_advance
[engine_bus]
engine_bus
fuel_rate
engine_controller
[air2fuel_ratio]
w
EGO
MAP
[engine_bus]
Engine
[can_bus]
transmission_controller
can_bus
pset
[transmission_bus]
transmission_bus
transmission_controller
_shaft
w_shaft
w_out
[transmission_bus]
Transmission and Torque Converter
[can_bus]
bus
brake_torque
brake_controller
[vehicle_speed]
[air2fuel_ratio]
air2fuel_ratio
Vehicle
100
throttle
0
brake
[vehicle_speed]
speed
[can_bus]
[transmission_bus]
transmission
gear
output
gear_velocities
[can_bus]
<speed>
vehicle_speed


Gravity
-9.81
Velocity
1
s
Velocity
Position
1
s
Position
Velocity
Position
[15]
IC
Position
-0.8
Elasticity


Velocity
20
10
0
-10
-20
-30
0
5
10
15
20
Position
20
15
10
5
0
0
5
10
15
20
Time offset: 0

Simulink™

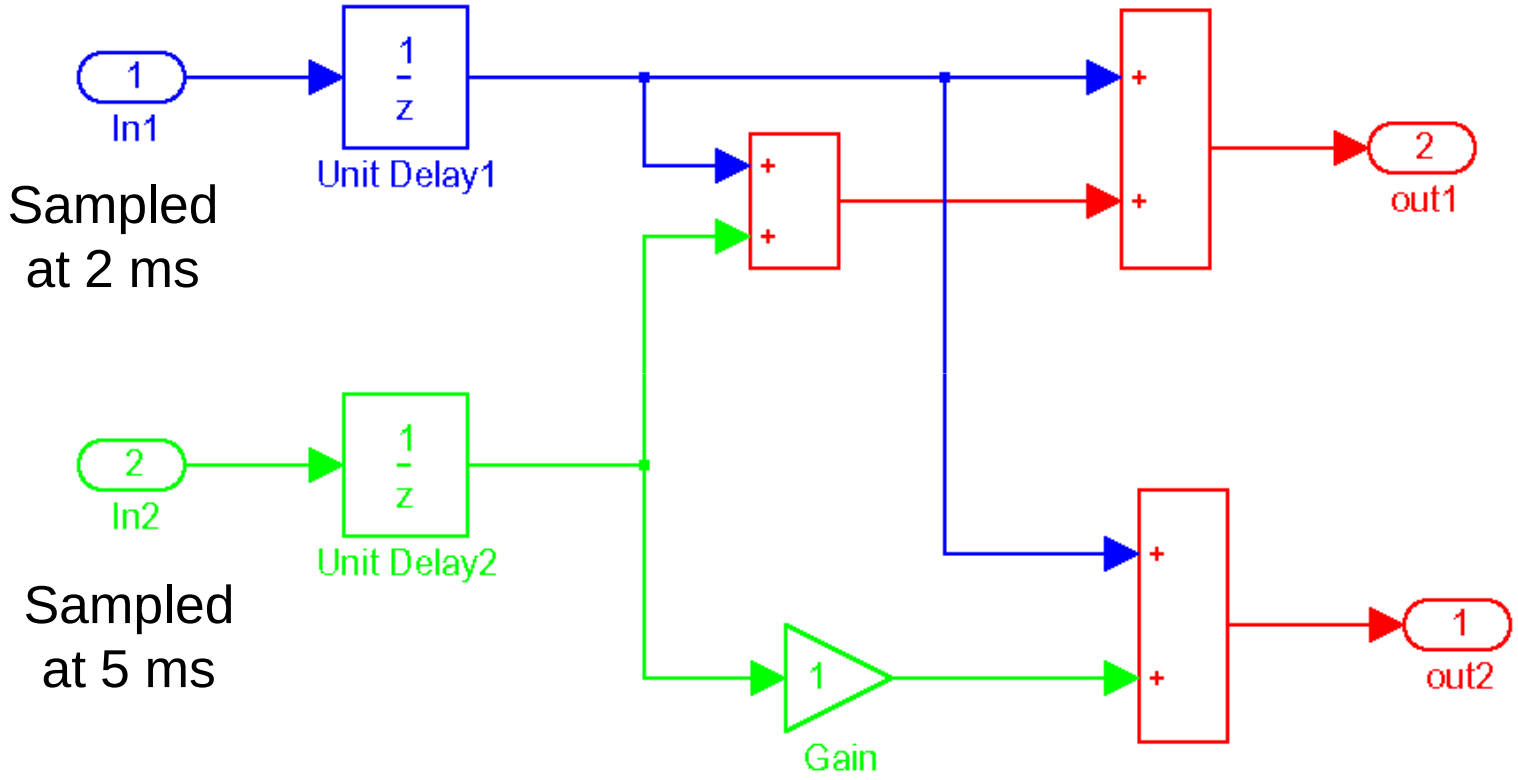
- Designed as a simulation tool, not a programming language
- No formal semantics
  - Depend on simulation parameters
  - No timing modularity
  - Typing depends on simulation parameters

We translate only discrete-time Simulink
(with no causality cycles)

From Simulink/Stateflow to Lustre

- Main issues:
  - Understand/formalize Simulink/Stateflow
  - Solve specific technical problems
    - Some are Lustre-specific, many are not
  - Implement
    - Keep up with The Mathworks' changes

A strange Simulink behavior



With Gain: model rejected by Simulink
Without Gain: model accepted!

Translating Simulink to Lustre

- 3 steps:
  - Type inference:
    - Find whether signal x is “real” or “integer” or “boolean”
  - Clock inference:
    - Find whether x is periodic (and its period/phase) or triggered/enabled
  - Block-by-block, bottom-up translation:
    - Translate basic blocks (adder, unit delay, transfer function, etc) as predefined Lustre nodes
    - Translate meta-blocks (subsystems) hierarchically

Simulink type system

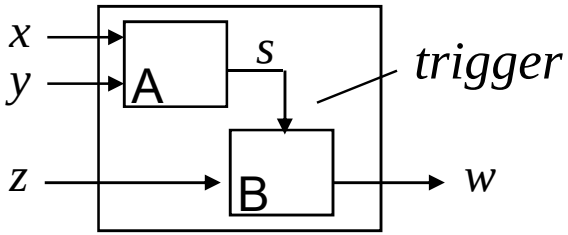
- Polymorphic types
  - "parametric" polymorphism (e.g., "Unit Delay" block)
  - "ad-hoc" polymorphism (e.g., "Adder" block)
- Basic block type signatures:

Constant$_\alpha$	α, $\alpha \in$ {double, single, int32, int16, …}
Adder	$\alpha \times \ldots \times \alpha \rightarrow \alpha$, $\alpha \in$ {double, …}
Relation	$\alpha \times \alpha \rightarrow$ *boolean*, $\alpha \in$ {double, …}
Logical Operator	*boolean* $\times \ldots \times$ *boolean* $\rightarrow$ *boolean*
Disc. Transfer Function	*double* $\rightarrow$ *double*
Unit Delay	$\alpha \rightarrow \alpha$
Data Type Converter$_\alpha$	$\beta \rightarrow \alpha$

- Type-inference algorithm: unification [Milner]
  - (In fact simpler since we have no terms)

Time in Simulink

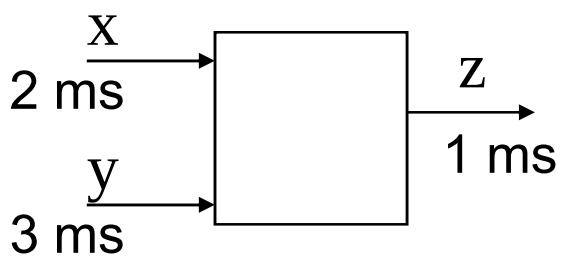
- Simulink has two timing mechanisms:
  - *sample times* : (period,phase)
    - Can be set in blocks: in-ports, UD, ZOH, DTF, …
    - Defines when output of block is updated.
    - Can be inherited from inputs or parent system.
  - *triggers (or "enables")* :
    - Set in subsystems
    - Defines when subsystem is "active" (outputs updated).
    - The sample times of all children blocks are inherited.



Simulink triggers
=
Lustre clocks

Sample times in Simulink

- Greatest-common divisor (GCD) rule :
  - A block fed with inputs with different rates:



- Other timing rules, e.g.:
  - Insert a unit delay when passing from a “slow” block to a “fast” block.

Formalization

- Sample time signatures of basic blocks:

$$
\begin{aligned}
\text{Output} &: \alpha \rightarrow \alpha, \alpha \in SampleTimes \\
\text{Math} &: \alpha_1 \times \cdots \times \alpha_n \rightarrow \textsf{gcd-rule}(\alpha_1, ..., \alpha_n), \alpha_i \in SampleTimes, i = 1, ..., n \\
\text{Switch} &: \alpha \times \beta \times \gamma \rightarrow \textsf{gcd-rule}(\alpha, \beta, \gamma), \alpha, \beta, \gamma \in SampleTimes \\
\text{Input} &: \alpha \rightarrow \alpha, \alpha \in SampleTimes \\
\text{Discrete}_\beta &: \alpha \rightarrow \alpha, \alpha \in SampleTimes, \beta = -1 \\
\text{Discrete}_\beta &: \alpha \rightarrow \beta, \alpha \in SampleTimes, \beta \in SampleTimes, \beta \neq -1 \\
\text{Triggered}_\alpha &: \alpha \times \cdots \times \alpha \rightarrow \alpha, \alpha \in SampleTimes \\
\text{Enabled}_\alpha &: \alpha \times \cdots \times \alpha \rightarrow \alpha, \alpha \in SampleTimes
\end{aligned}
$$

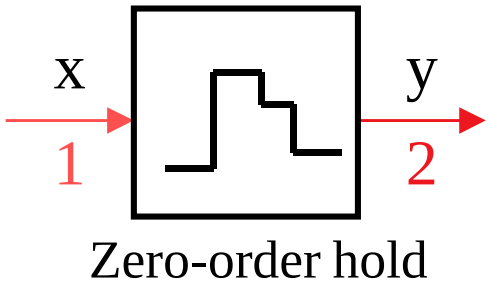
Sample time inference algorithm

- Sample times = types = terms:
  - α (unknown)
  - (period,phase) constants, e.g.: (1, 0), (2, 1), etc
  - GCD(t_1, t_2)
- Terms simplify to a canonical form
  - GCD(β, (2,0), (3,0), α) $\rightarrow$ GCD((1,0), α, β)
- Term unification, e.g. :
  - From the equations: z = GCD(x,y) and x = z
  - We get: x = GCD(x, y)
  - Thus: x = GCD(y)
  - Thus: x = y = z

Overview of clock inference algorithm

- Infer the sample time of every Simulink signal.
- Check Simulink's timing rules.
- Create Lustre clocks for Simulink sample times and triggers.
  - Basic clock: GCD of all sample times, e.g., 1ms.
  - Other clocks: multiples of basic clock, e.g.
  - *true false true false* $\cdots$ = 2ms.

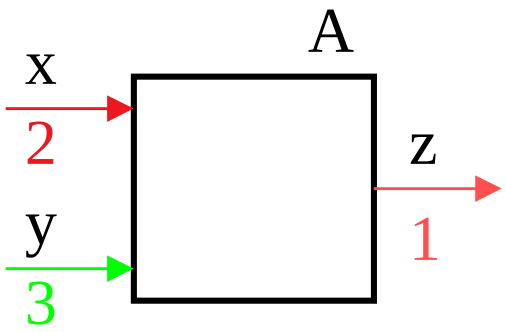
From Simulink sample times to Lustre clocks



Zero-order hold

```
cl_1_2 = make_cl_1_2();
y = x when cl_1_2;
```

cl_1_2 = {*true*, *false*, *true*, *false*…}



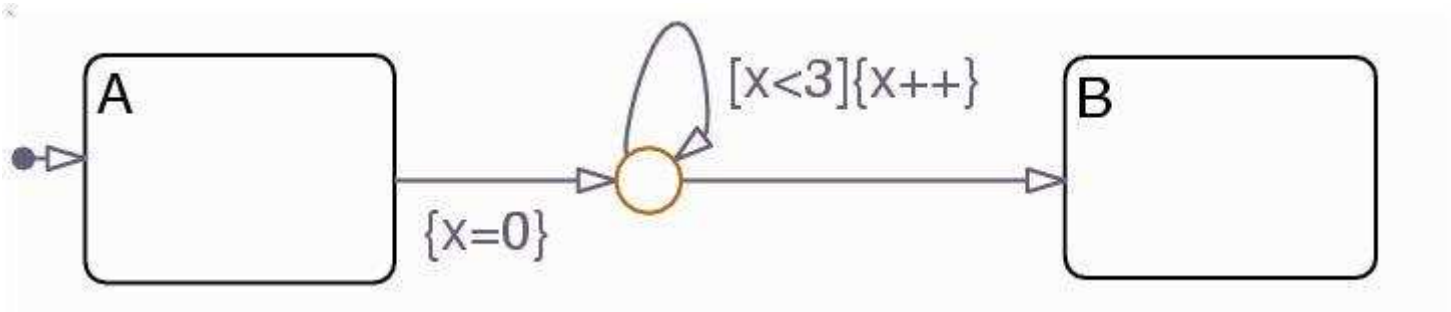

```
xc = current(x);
yc = current(y);
z  = A(xc, yc);
```

Stateflow

- Main problem: “unsafe” features
  - Non-termination of simulation cycle
  - Stack overflow
  - Backtracking without “undo”
  - Semantics depends on graphical layout
  - Other problems:
    - “early return logic”: returning to an invalid state
    - Inter-level transitions
    - …

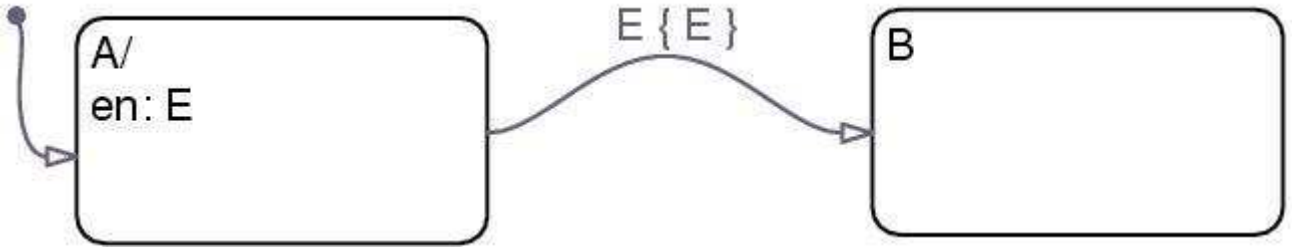
Stateflow problems: non-terminating loops

- Junction networks:

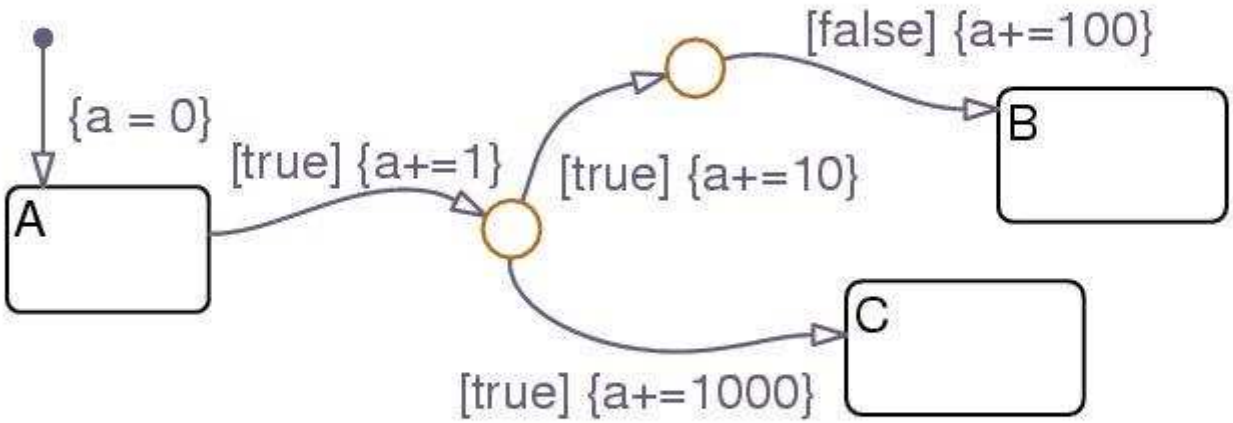


Stateflow problems: stack overflow

- When event is broadcast:
  - Recursion and run-to-completion
- Stack overflow:

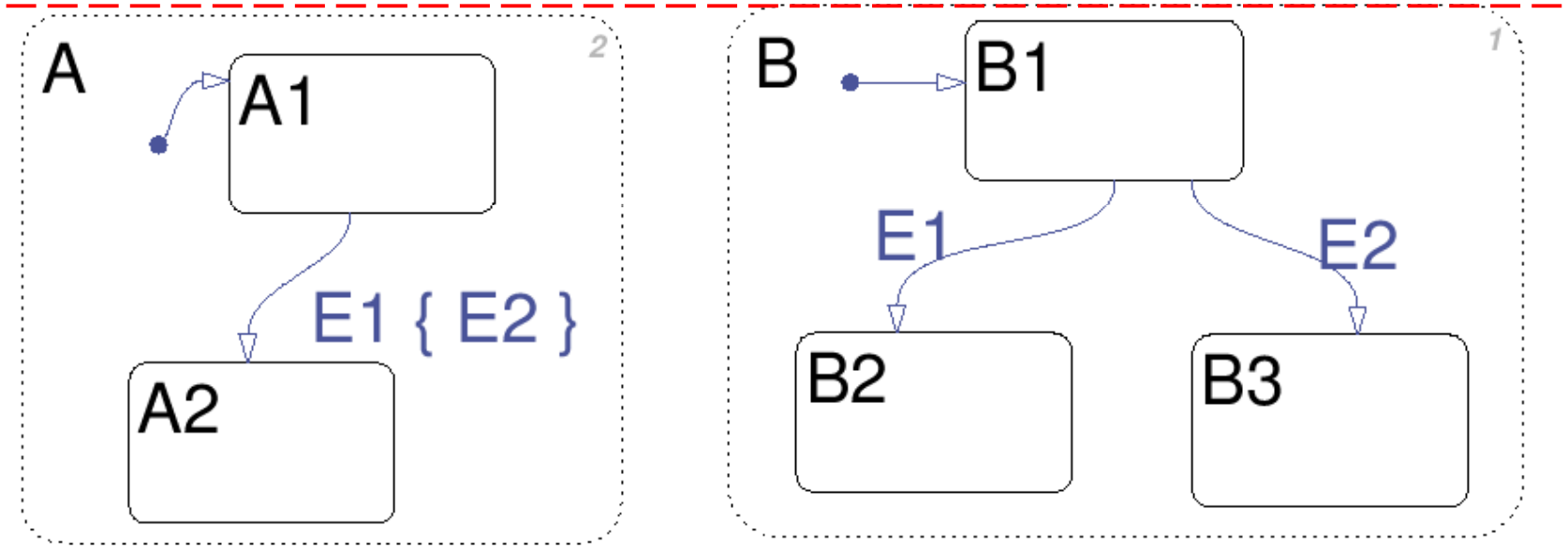


Stateflow problems: backtracking without “undo”



Stateflow problems: semantics depends on layout

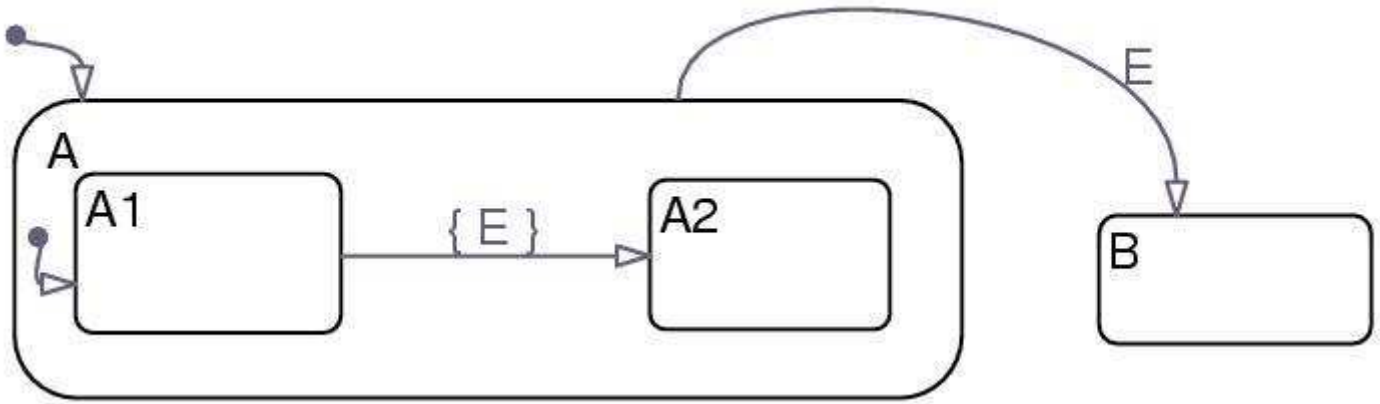
- “top-to-bottom, left-to-right” rule for states:



- “12 o’clock” rule for transitions

Stateflow problems: “early return logic”

- Return to a non-active state:



A “safe” subset of Stateflow

- Safe = terminating, bounded-memory, “clean”
- Problem undecidable in general
- Different levels of “safeness”:
  - Static checks (cheap but strict)
  - Dynamic verification (heavy but less strict)

A statically safe subset of Stateflow

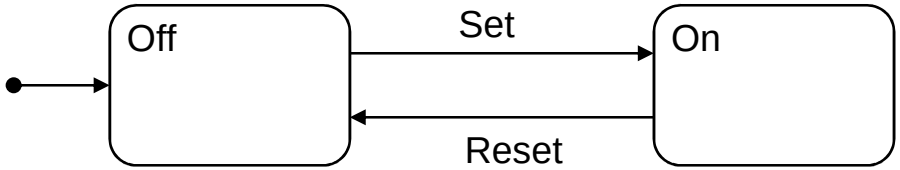
- Static checks include:
  - Absence of multi-segment loops
  - Acyclicity of triggering/emitted events
  - No assignments in intermediate segments
  - Outgoing junction conditions form a cover (implies no deadlocks)
  - Outgoing junction conditions are disjoint (implies determinism)

From Stateflow to Lustre

- Main difficulty:
  - Translating state-machines into dataflow
- Approach:
  - Encode states with Boolean variables
  - Encode execution order by “dummy” dependencies

Translation to Lustre

- Encoding of states and events as boolean flows
- “mono-clock”




```
node SetReset0(Set, Reset: bool)
returns (sOff, sOn: bool);
let
  sOff = true ->
    if pre sOff and Set then false
    else if (pre sOn and Reset) then true
    else pre sOff;
  sOn = false ->
    if pre sOn and Reset then false
    else if (pre sOff and Set) then true
    else pre sOn;
tel
```

Readings from the Verimag group:

- Overall approach:
  - http://www-verimag.imag.fr/~tripakis/papers/lctes03.ps
- Simulink to Lustre:
  - http://www-verimag.imag.fr/~tripakis/papers/acm-tecs.pdf
- Stateflow to Lustre:
  - http://www-verimag.imag.fr/~tripakis/papers/emsoft04.pdf
- Multi-task implementations:
  - http://www-verimag.imag.fr/~tripakis/papers/acm-tecs07.pdf
  - http://www-verimag.imag.fr/TR/TR-2004-12.pdf
  - http://www-verimag.imag.fr/~tripakis/papers/emsoft05.pdf
  - http://www-verimag.imag.fr/~tripakis/papers/emsoft06.pdf
- Adrian's thesis:
  - http://www-verimag.imag.fr/~curic/thesis_AdrianC_11_25.pdf
- Christos' thesis:
  - http://www-verimag.imag.fr/~sofronis/sofronis-phd.pdf
- A tutorial chapter on synchronous programming:
  - http://www-verimag.imag.fr/~tripakis/papers/handbook07.pdf

End