

Towards reusable real-time objects

by Brian Nielson and Gul Agha 1999

Concurrency Seminar
SS 2005

Speaker: Martin Bättig

Summary

- Problem:
 - large and complex real-time systems
 - component-based development preferred
 - hardly reusable code (embedded constraints)
- Solution:
 - separate constraints and functional behaviour
- Technologie:
 - actor model
 - RT-Synchronizers[™] language

Outline

- Problem discussion
- Separation and reuse
- Model of Separation
- Actor Model
- RT-Synchronizers⁻
- Formal Definition
- Middleware Scheduling
- Conclusion

Problem

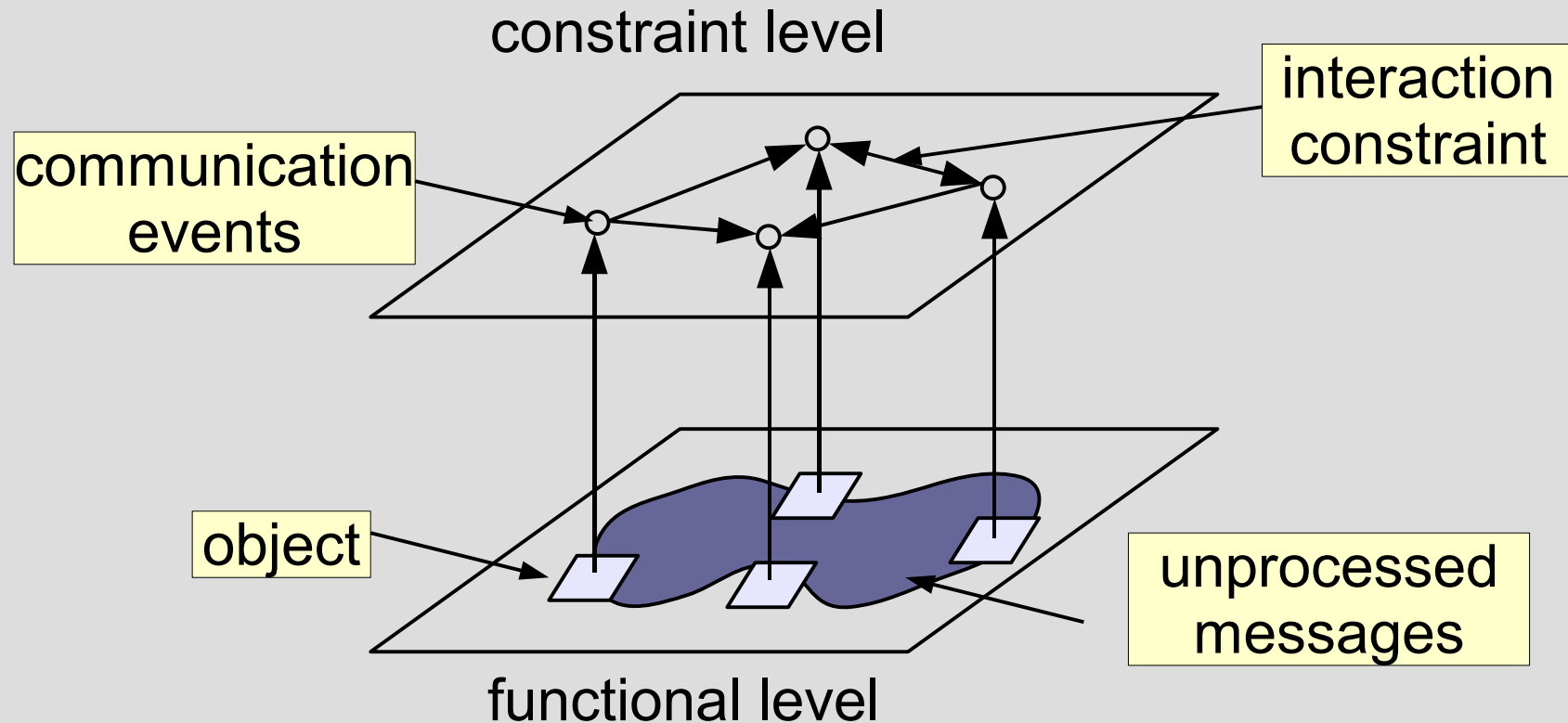
- real-time systems can be large and complex
- typical usage monitor and regulate devices
- safety critical applications
- need strict end-to-end time constraints

- benefit from component-based development
- often embedded real-time constraints
 - => interdependent objects
 - => reusing real-time components problematic

Separation and reuse

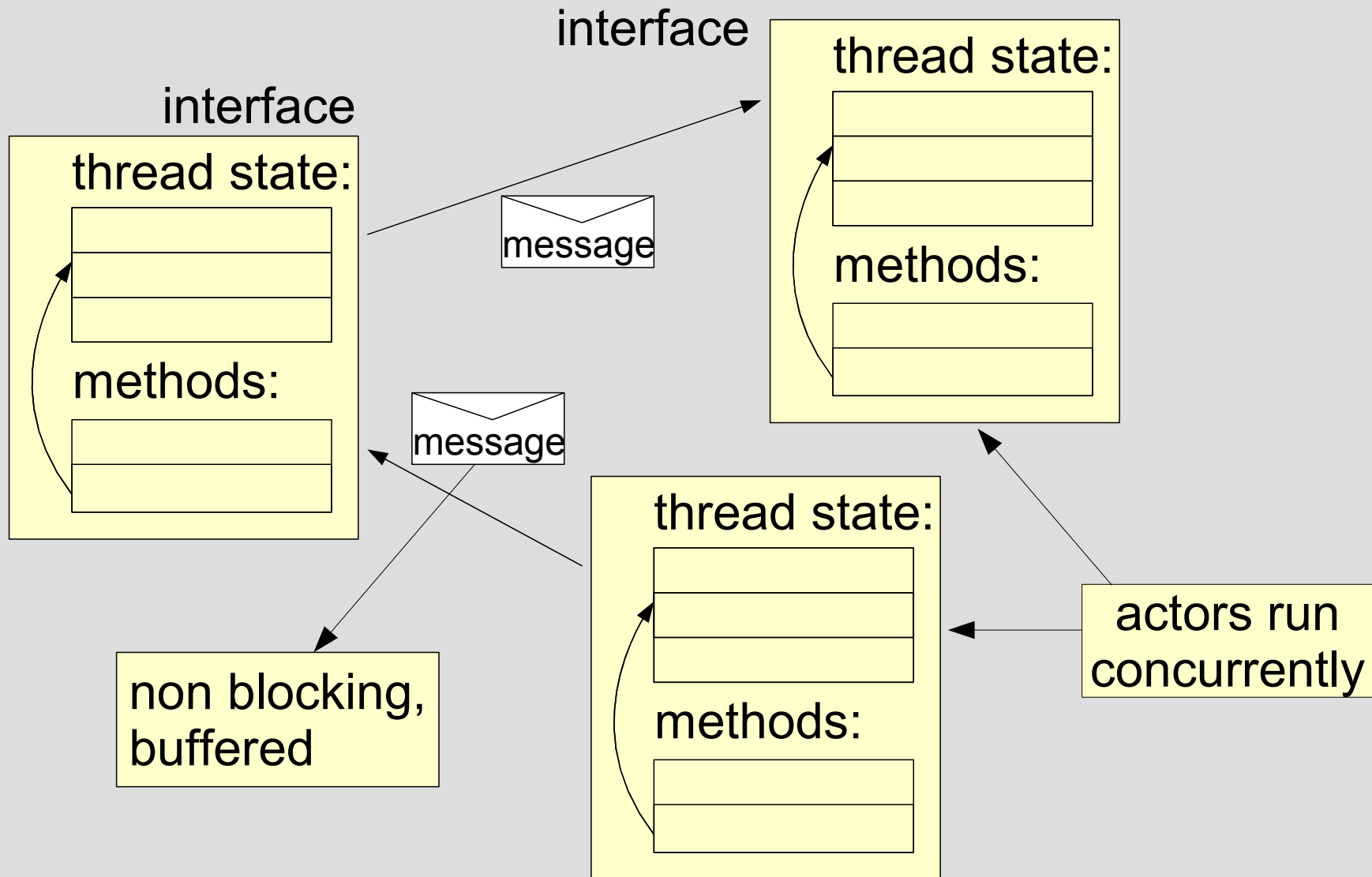
- real-time properties are likely to differ between applications
- embedded properties force to retest an object after a change
- restrictions on real-time objects
 - no self scheduling
 - no hardwired synchronization constraints
- real-time properties are global
- constraints are a source of reuse
=> factor out common constraints

Separation Model



interaction constraints = real-time and synchronization constraints

Actor Model



Actor Model: Messages

- each actor identified by actor reference (mail address)
- message sending: **send** a.m(pv)
 - a = actor reference
 - m = method to be invoked
 - pv = values to be communicated
- language unspecified

Actor Model: Example (Boiler)

```
actor pressureSensor ( ) {
  real value;
  method read(actorRef customer) {
    send customer.reading(value);
  }
}
actor steamValve ( ) { ... } // unspecified
actor controller (actorRef sensor, valve) {
  method loop( ) {
    send self.loop( );
    send sensor.read(self); } } asynchronous
  method reading(real pressure) {
    newValvePos=computeValvePos(pressure);
    send valve.move(newValvePos);
  }
}
```

RT-Synchronizers⁻ : Constraints

- language used to express constraints
- constraints enforced on messages
- p_1, p_2 : message patterns
form: $x_1(x_2)$ when b (b is a guard over x_2)
example: heater.stop(temp) when temp < 10
- constraints:
 $p_1 \Rightarrow p_2 \prec y$: demand for p_2 in y time units
 $p_1 \Rightarrow p_2 \succ y$: y time units must pass before p_2
- no constraints until a message pattern matches

RT-Synchronizers⁻: Structure

e.g. `int i = 1;`

synchronizer (a_1, \dots, a_n) {

State Declaration

instantiation
parameters

$p_{11} \Rightarrow p_{21} \sim y_1$

\vdots

$p_{1n} \Rightarrow p_{2n} \sim y_n$

Constraints

$\sim \in \{>, <\}$.

Triggers

$p_1 : \bar{x} := \overline{exp}$

\vdots

$p_k : \bar{x} := \overline{exp}$

}

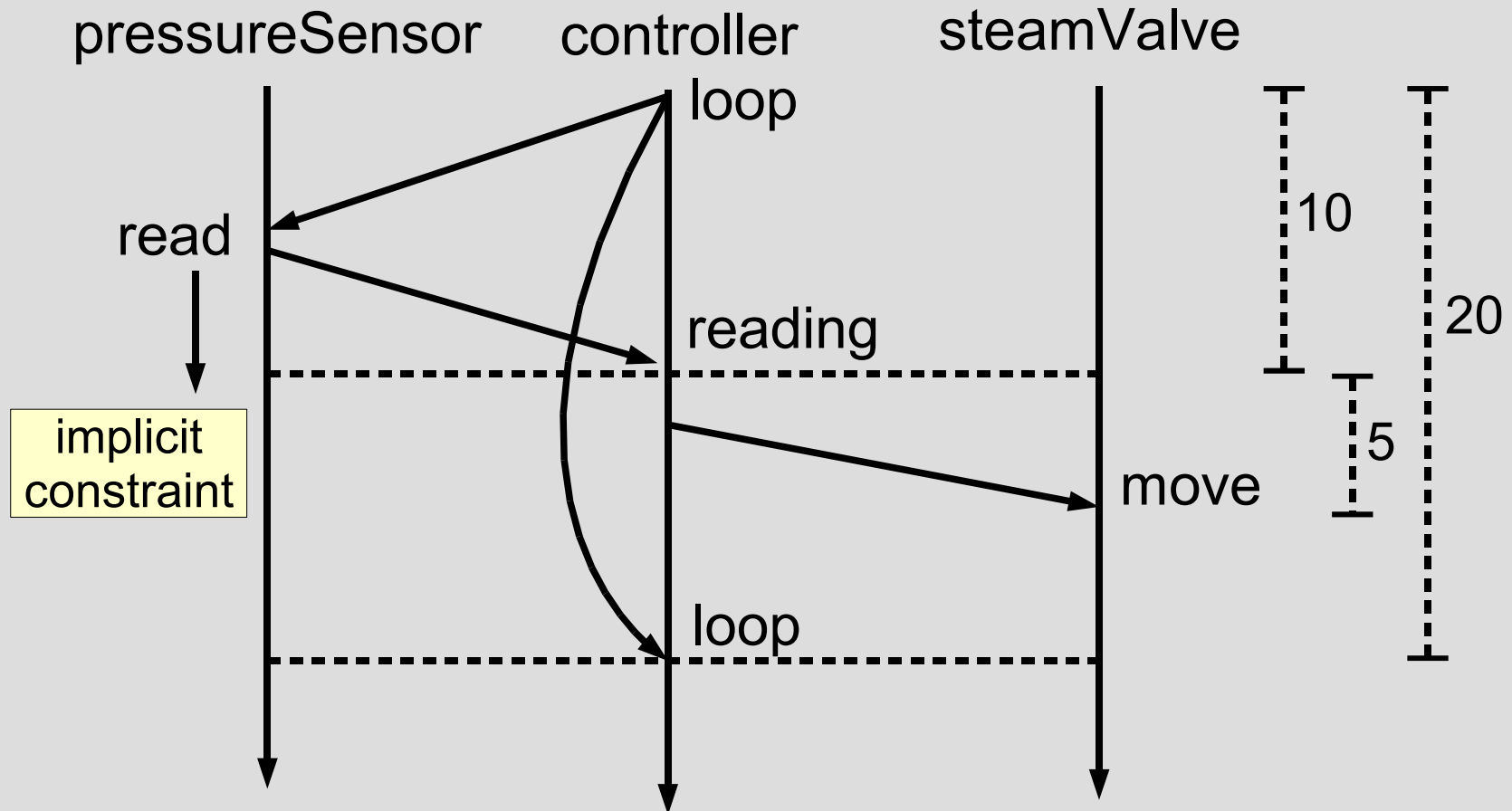
Synchronizers
act concurrently

RT-Synchronizers⁻: Example (1)

```
actor pressureSensor ( ) { ... };
actor steamValve ( ) { ... };
actor controller (actorRef sensor, valve) { ... };

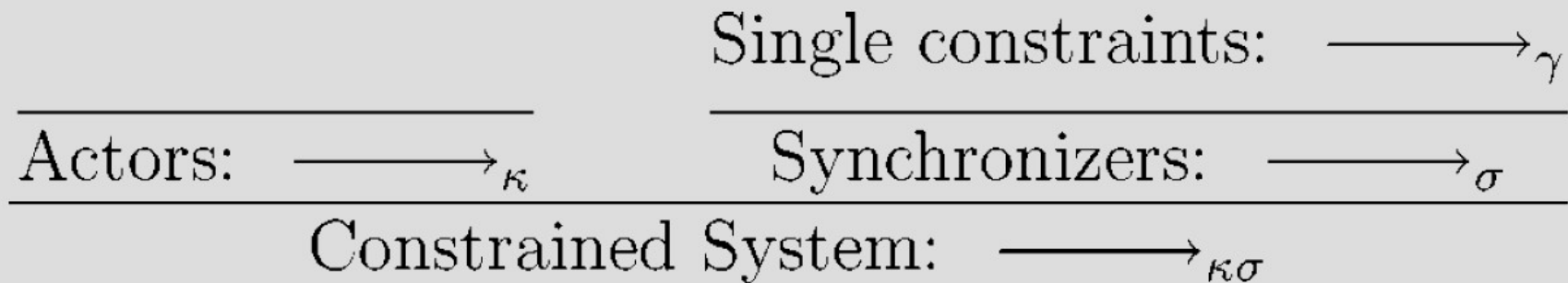
synchronizer boilerConstraints (actorRef: controller, valve) {
  //periodic loop:
  controller.loop ⇒ controller.loop < 20+ε }
  controller.loop ⇒ controller.loop > 20-ε } Periodic
  //deadline on reading:
  controller.loop ⇒ controller.reading < 10
  //deadline on move:
  controller.reading ⇒ valve.move < 5
}
```

RT-Synchronizers: Example (2)



Formal Definition (1)

- formal definition of the model
- separated transition systems for both the Actors and RT-Synchronizers
- transition systems are put into parallel



- transition sequence: one possible schedule

Formal Definition (2)

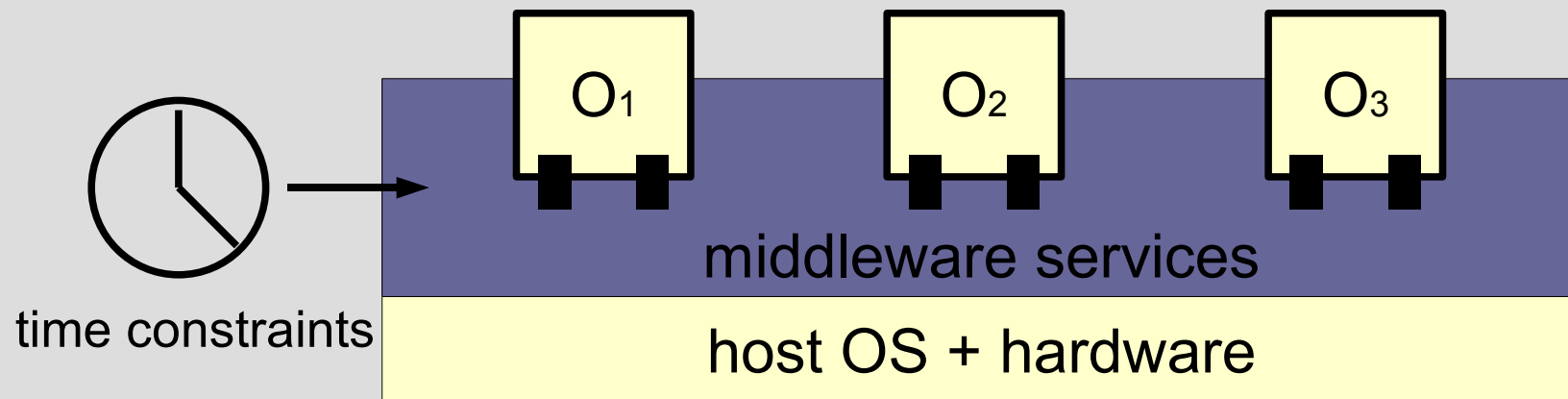
- system has the ability to let time pass:

$$\langle \alpha \mid \mu \rangle \xrightarrow{\varepsilon(d)}_{\kappa} \langle \alpha \mid \mu \rangle$$

- time is not allowed to pass if a constraint fails
- time is not required to pass between two events
- time lock: unsatisfiable deadline constraints
=> no time progress possible
- cluster point: bounded interval of time in which an infinite number of events occur
- compiler should warn about unsatisfiable constraints

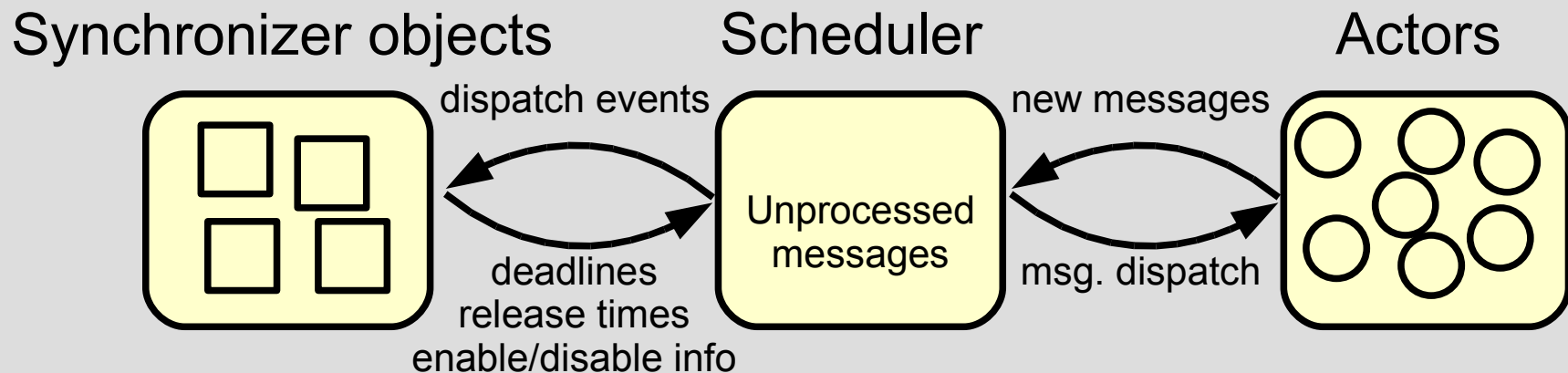
Middleware scheduling

- idea: Use a middleware scheduling/event dispatching service
- application consisting of two parts, objects and time constraints
- service schedules messages according to constraints given by the synchronizers



Middleware scheduling: Constraint directed scheduling

- maintain synchronizer objects at run time
- scheduler uses the information to assign deadlines and release times
- use time-based scheduling e.g. Earliest-Deadline-First



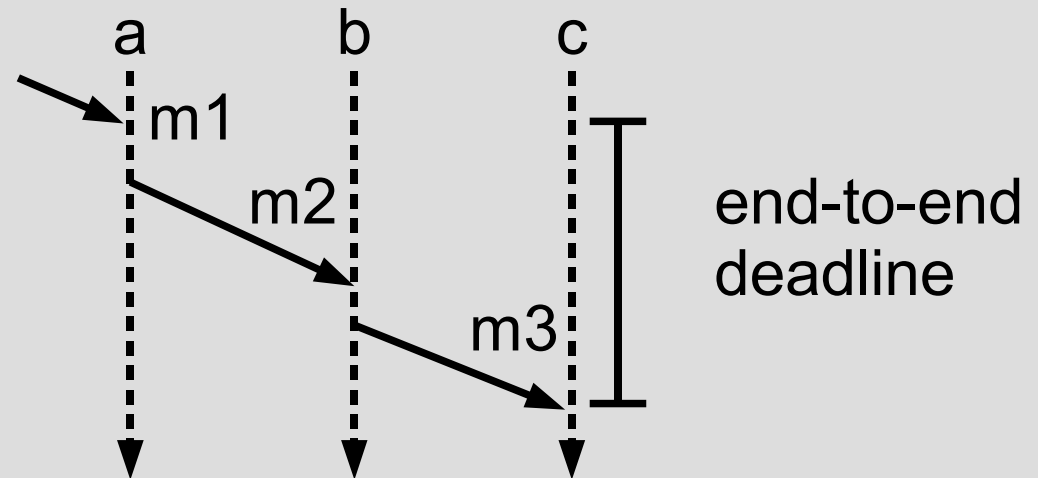
- centralized vs. distributed synchronizers

Middleware scheduling: Constraint propagation

- we have end-to-end timing constraints
=> derive intermediate deadlines

- actors: a, b, c
- messages: $m\{1-3\}$
- constraint:

$$a_{m1} \Rightarrow c_{m3} < 10$$



- heuristic function of slack time and method computation time
- include call graph with worst case execution time

Conclusion

- possibility to separate functional behavior and real-time constraints
- formulated in context of the Actors and the new introduced RT-Synchronizers⁻ language
- defined a semantic for the model
- strategy for implementing soft real-time systems

Questions

