McGill

ECSE 421 Lecture 2: Specifying Requirements Early Design Stages

ESD Chapter 2

Last Time

- Introduction to Embedded System Design
  - Embedded System
  - Cyber-physical system
  - Real-time system
- Common characteristics
- Problem: *design challenges*
- Solution: *design flow*

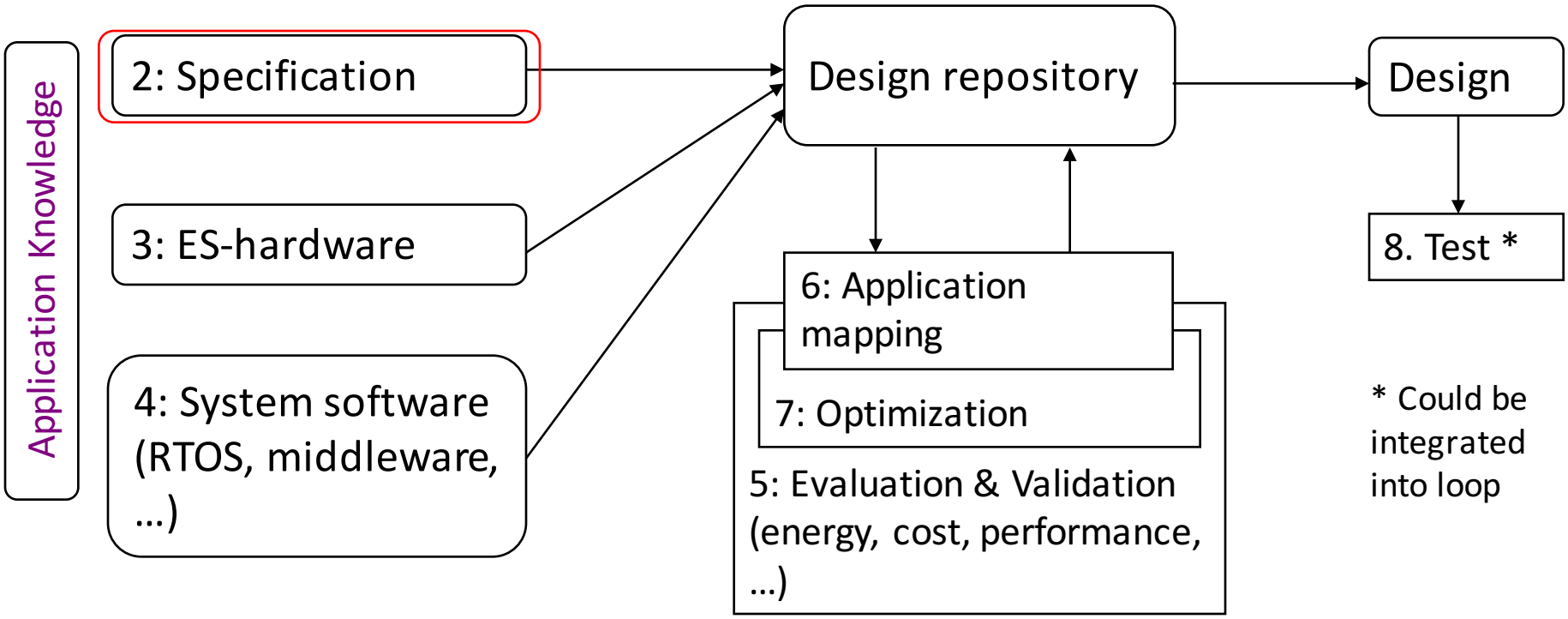
Where Are We?

W	D	Date		Topic	ESD	Out	In	Notes
1	T	12-Jan-2016	L01	Introduction to Embedded System Design	1.1-1.4			
	R	14-Jan-2016	L02	Specifying Requirements / MoCs / MSC	2.1-2.3			
2	T	19-Jan-2016	L03	CFSMs	2.4			
	R	21-Jan-2016	L04	Data Flow Modeling	2.5			
3	T	26-Jan-2016	L05	Petri Nets	2.6			
	R	28-Jan-2016	L06	Discrete Event Models	2.7			
4	T	2-Feb-2016	L07	DES / Von Neumann Model of Computation	2.8-2.10			Guest: Z. Al-bayati
	R	4-Feb-2016	L08	Sensors	3.1-3.2			
5	T	9-Feb-2016	L09	Processing Elements	3.3			
	R	11-Feb-2016		Processing Elements	3.3			
6	T	16-Feb-2016	L10	More Processing Elements / FPGAs				
	R	18-Feb-2016	L11	Memories, Communication, Output	3.4-3.6			
7	T	23-Feb-2016	L12	Embedded Operating Systems	4.1			
	R	25-Feb-2016		***Midterm exam: in-class, closed book***				Chapters 1-3
8	T	2-Mar-2016		***No class***				Winter break
	R	4-Mar-2016		***No class***				Winter break
	T	9-Mar-2016	L13	Middleware	4.4-4.5			
	R	11-Mar-2016	L14	Performance Evaluation	5.1-5.2			
9	T	16-Mar-2016	L15	More Evaluation and Validation	5.3-5.8			
	R	18-Mar-2016	L16	Introduction to Scheduling	6.1-6.2.2			
10	T	23-Mar-2016	L17	Scheduling Aperiodic Tasks	6.2.3-6.2.4			
	R	25-Mar-2016	L18	Scheduling Periodic Tasks	6.2.5-6.2.6			

Today

- Requirements $\Rightarrow$ Specification
  - What do we need to specify?
  - How can we specify it?
- Models of Computation
  - What does it mean to "compute"?
  - Is there more than one way to describe "computation"?
- Modeling in early design stages

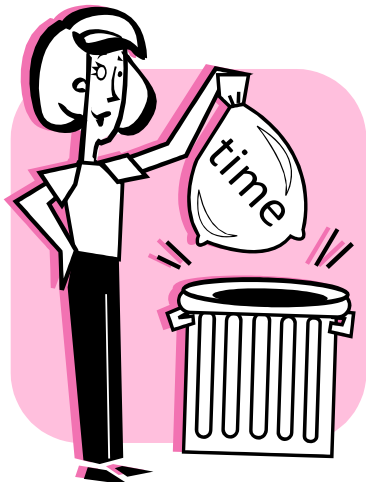
Hypothetical Design Flow



Numbers denote sequence of chapters

System Specifications

- Why bother with specification?
- If there's an error in the specification
  - It will be difficult to get the design right
  - This potentially wastes a lot of time
- Typically, we work with **models** of the **system under design** (SUD)
- What is a model anyway?



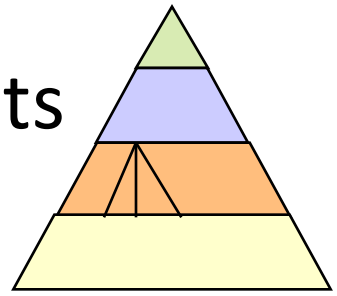
Models

Definition: *A* model *is a simplification of another entity … The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task.* [Jantsch, 2004]

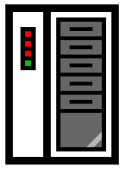
What requirements do we have for our models?

Spec. Req's: Hierarchy

- Humans incapable of understanding systems with more than ~5 objects
- Most actual systems require more objects
  ⇒ Hierarchy
  - Behavioral hierarchy
    Examples: states, processes, procedures
  - Structural hierarchy
    Examples: processors, racks, printed circuit boards

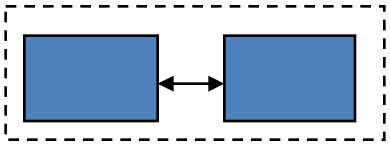


proc
  proc
    proc



Spec. Req's: Component-based design

- Systems must be designed from components
- Must be "easy" to derive system behavior from the behavior of subsystems
- Must provide support for
  - Concurrency
  - Synchronization and communication
- Work of Sifakis, Thiele, Ernst, …



Spec. Req's: Timing

- Modeling application timing behavior is essential for embedded and cy-phy systems!
  - Additional information (periods, dependences, scenarios, use cases) is helpful, too!

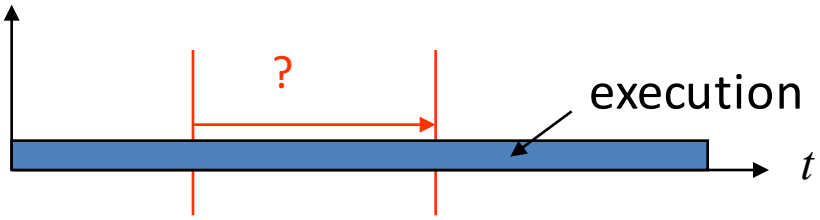
- Furthermore, the performance of the underlying platform must be known
  - And often must be predictable
- This has far-reaching consequences for design processes!

"The lack of timing in the core abstraction (of computer science) *is a flaw, from the perspective of embedded software"* [Lee, 2005]

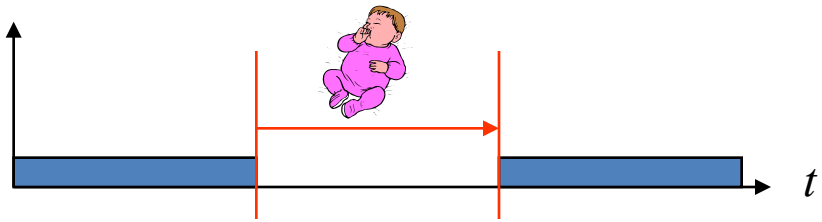
Specifying Timing

- Four types of techniques are required for specifying timing requirements [Burns, 1990]

1. Measure elapsed time
   E.g., to check, how much time has elapsed since last call

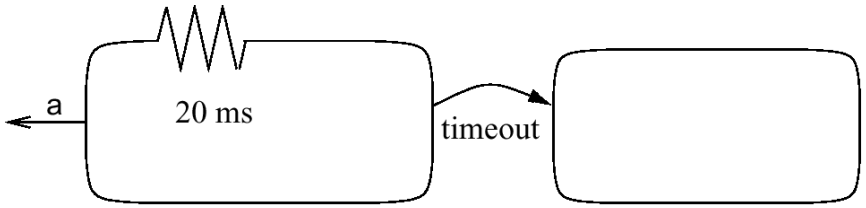


2. Delay processes

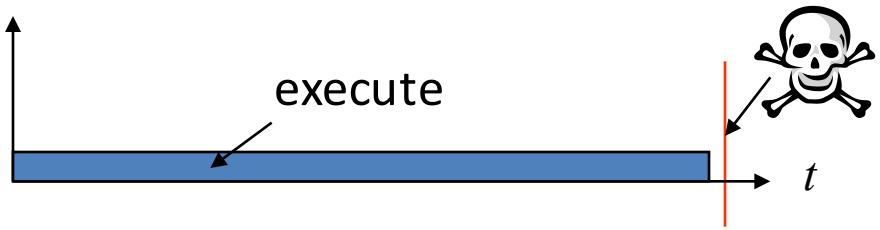


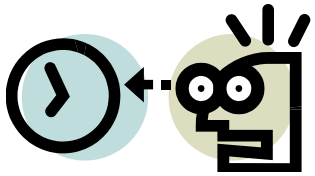
Specifying Timing, Cont'd

3. Specify timeouts
   E.g., to stay in a certain state a maximum time



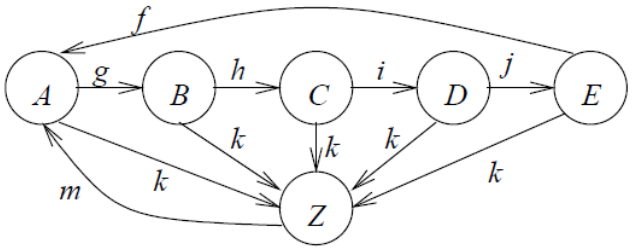
4. Specifying deadlines and schedules





Spec. Req's: Supporting Reactive Systems

- *State-oriented behavior*
  - Required for reactive systems; classical automata are insufficient
- *Event-handling*
  - External or internal events
- *Exception-oriented behavior*
  - Describing exceptions for every state is inefficient



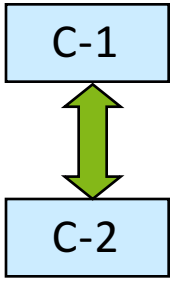
We will see how all the arrows labeled k can be replaced by a single one.

Additional Specification Requirements

- Presence of programming elements
- Executability (no algebraic specification)
- Support for the design of large systems ($\Rightarrow$ OO)
- Domain-specific support
- Readability
- Portability and flexibility
- Termination
- Support for non-standard I/O devices
- Non-functional properties
- Support for the design of dependable systems
- No obstacles for efficient implementation
- Adequate model of computation

Models of Computation

- **What does it mean "to compute"?**
- Models of computation define:
  - Components and an execution model for computations for each component
  - Communication model for exchange of information between components



Why Not von Neumann?

- Data races occur when accessing shared resources

```
task a {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

```
task b {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

Race-free access to shared memory protected by S is possible

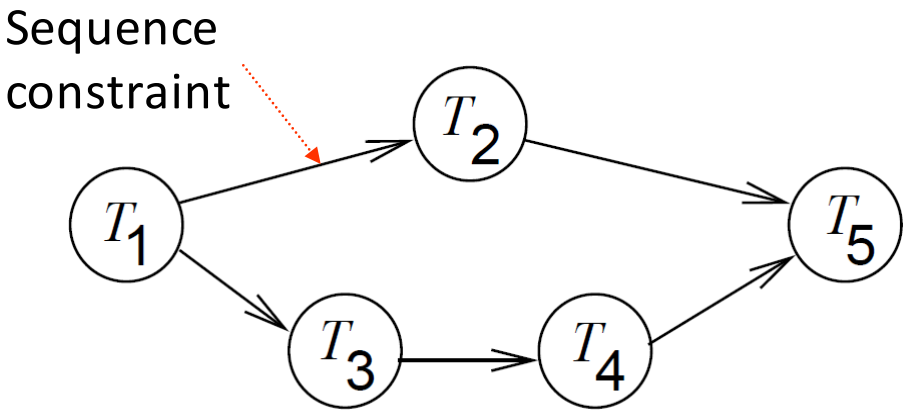
- In general-purpose computing, we address this
  - With synchronization primitives (semaphores)
  - With cache coherence protocols

Challenges With the von Neumann MoC

- Thread-based multiprocessing may access global variables
- Operating systems theory tells us that
  - access to global variables may lead to race conditions,
  - to avoid these, we need to use mutual exclusion,
  - mutual exclusion may lead to deadlocks, and
  - avoiding deadlocks is possible only if we accept performance penalties

Avoiding deadlock is hard in practice; more on this later

Dependence Graphs: Definition

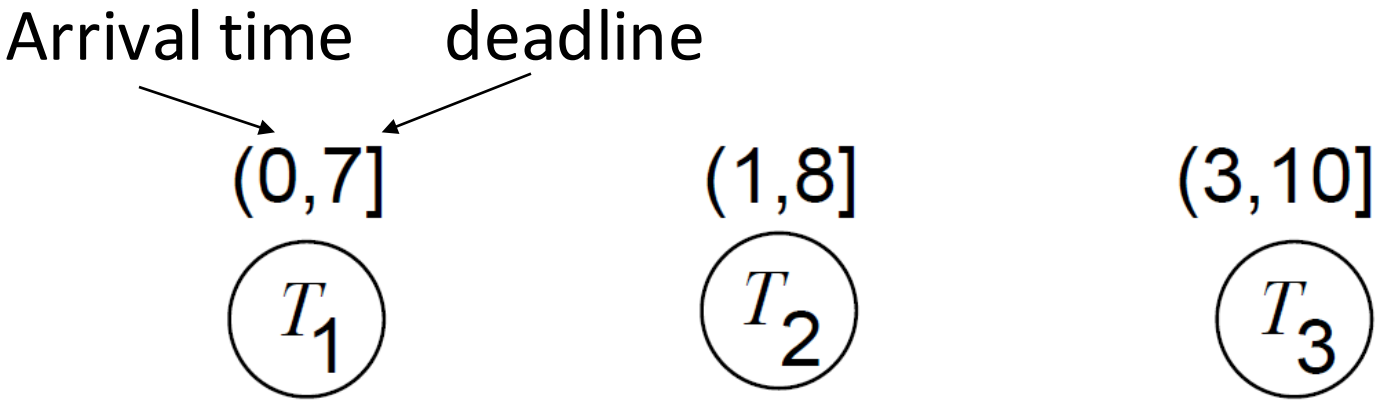


Nodes could be programs or simple operations

- **Def.:** A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a relation.
- If $(v_1, v_2) \in E$, then v_1 is called an **immediate predecessor** of v_2 and v_2 is called an **immediate successor** of v_1.
- Suppose E^* is the transitive closure of E.
  If $(v_1, v_2) \in E^*$, then v_1 is called a **predecessor** of v_2 and v_2 is called a **successor** of v_1.

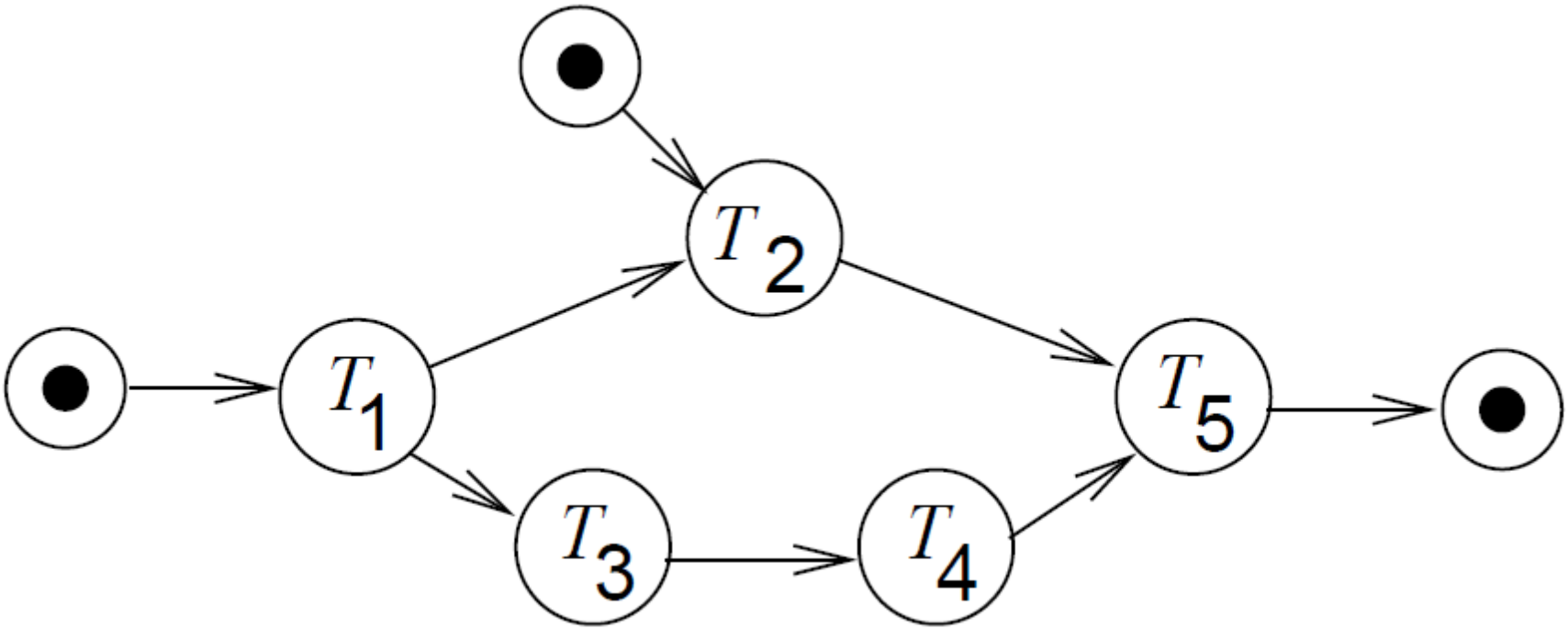
Dependence Graphs: Timing Information

- Dependence graphs may contain additional information, for example, timing information



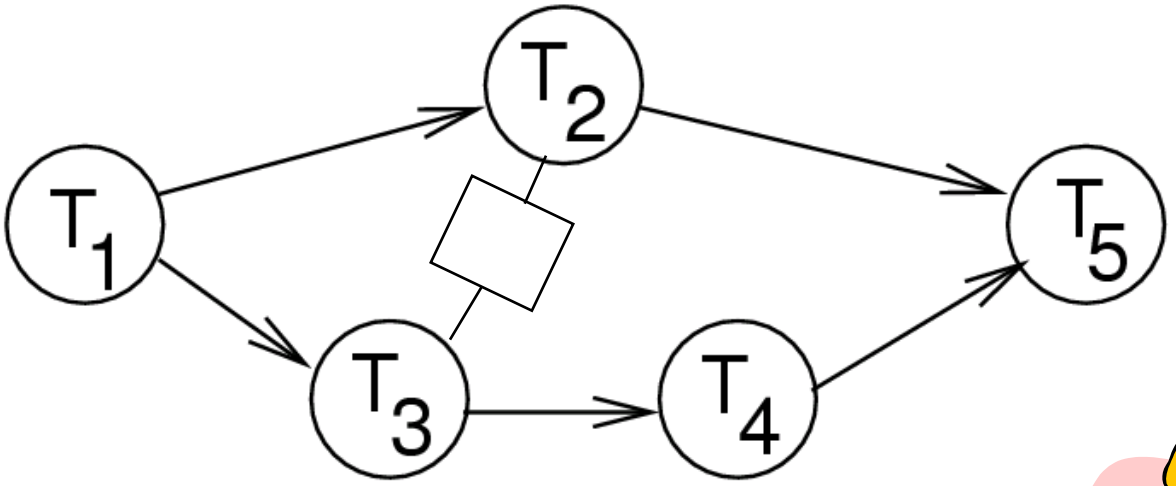
Dependence Graphs: I/O-information

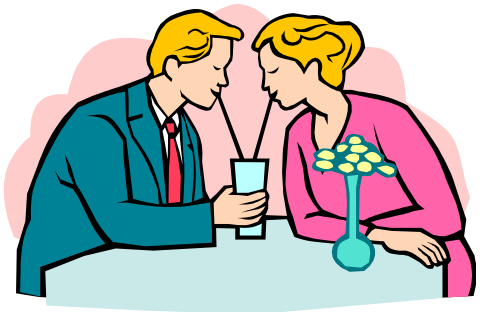
- Nodes can also be used to capture information about inputs and outputs



Dependence Graphs: Shared Resources

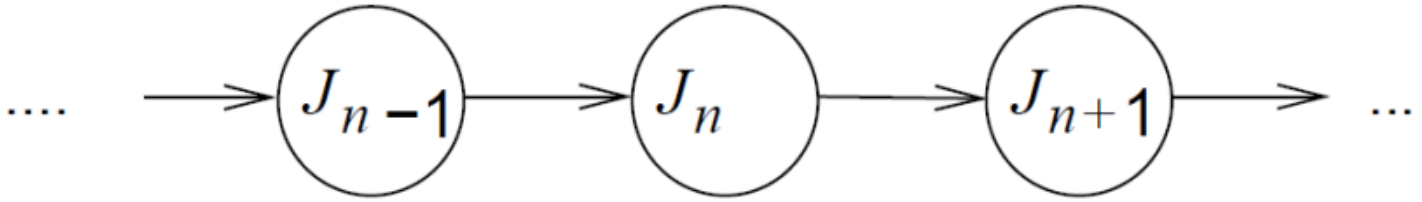
- Additional information can capture which tasks use shared resources (i.e., structural hazards)





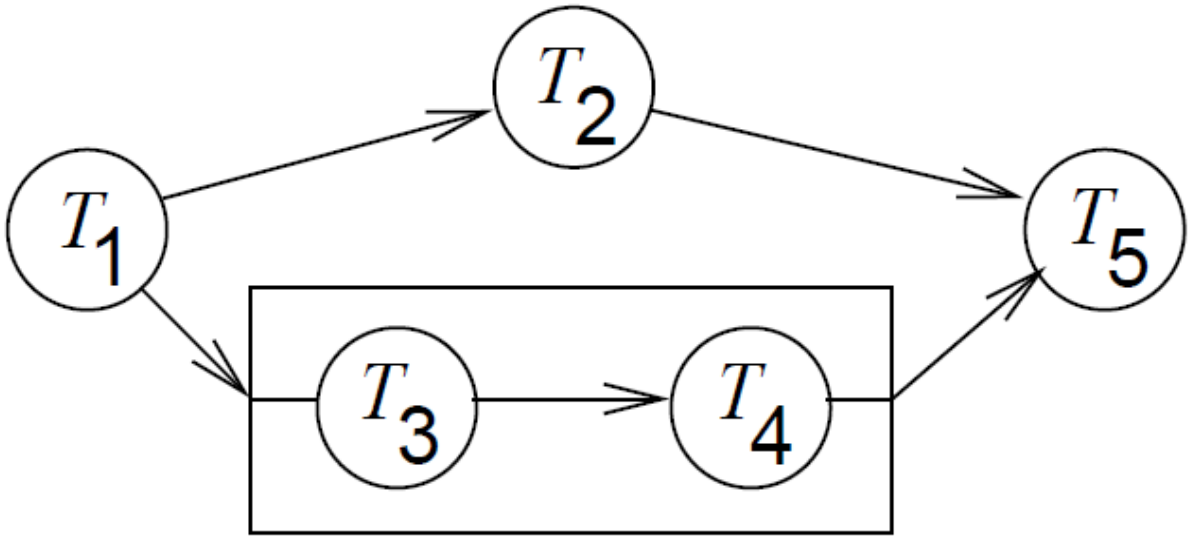
Dependence Graphs: Periodic Schedules

- A job is a single execution of the dependence graph
- Periodic dependence graphs are infinite



Dependence Graphs: Hierarchy

- Dependence graphs can be hierarchical
- This makes it easier to reason about complex graphs both for
  - Human developers, and
  - Algorithms



Modeling Communication

- Two basic approaches
  - Shared memory
  - Message passing
- Shared memory
  - Variables are accessible to several components/tasks

Shared Memory

- Data races are possible
  - Leading to potentially inconsistent results
- Exclusive access to critical sections must be guaranteed

```
task a {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

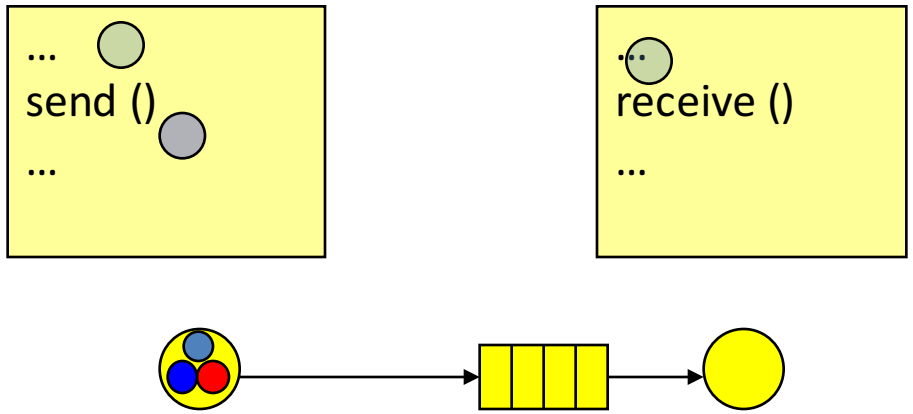
```
task b {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

Race-free access to shared memory protected by S is possible

P(S) and V(S) are **semaphore** operations,
allowing at most n accesses, n =1 in this case (mutex, lock)

Non-blocking Message Passing

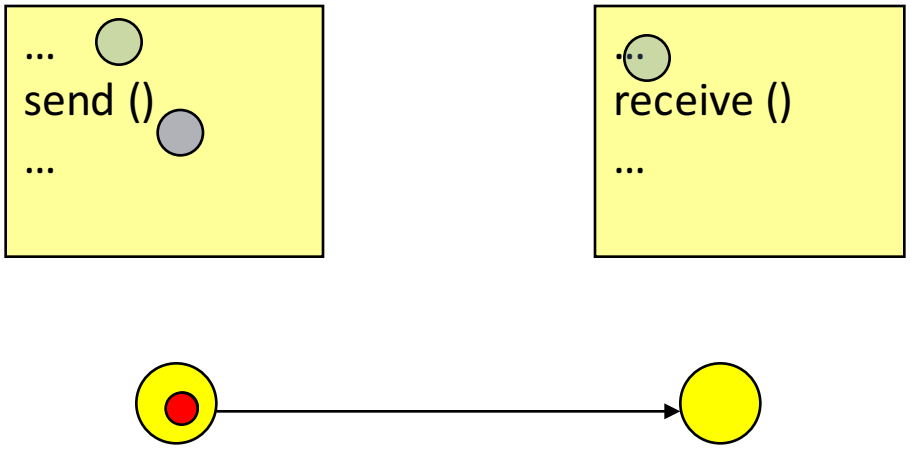
- Asynchronous message passing
- Sender does not have to wait until the message has arrived



- Potential problem: buffer overflow

Blocking Message Passing

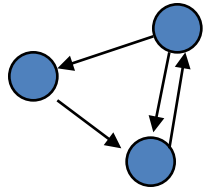
- Synchronous message passing; rendez-vous
- Sender will wait until receiver has received the message



- No buffer overflow, but reduced performance

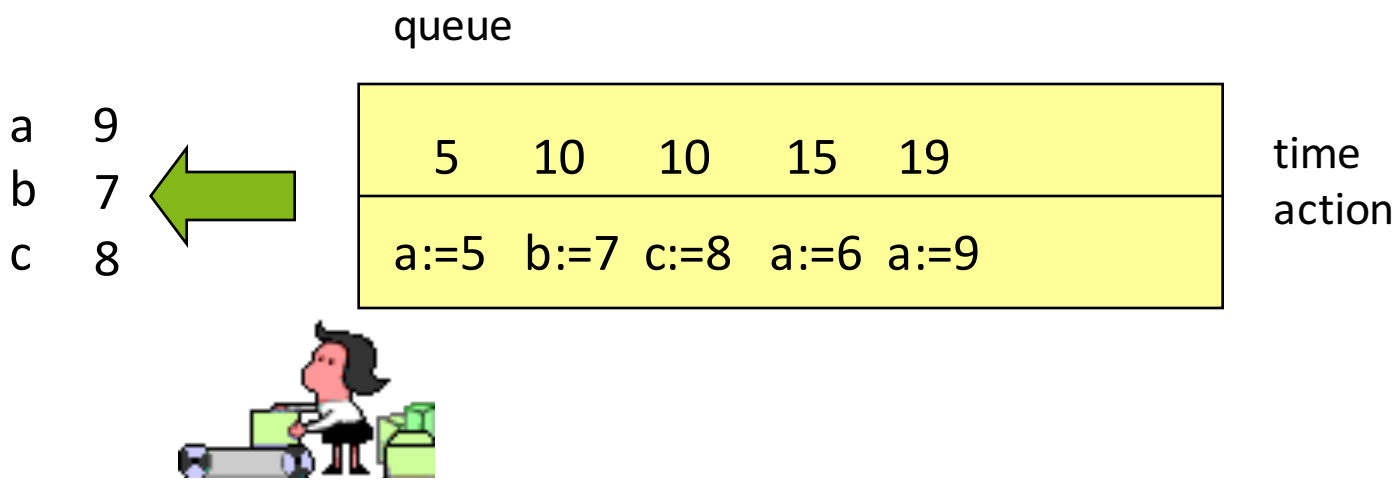
Organizing Computation

- Von Neumann
  - Sequential execution, program memory, etc.
- Finite state machines
  - Model the flow of execution
- Data flow
  - Models the flow of data
- Petri nets
  - Model synchronization



Organizing Computation, Cont'd

- Discrete event simulation



- Differential equations

$$\frac{\partial^2 x}{\partial t^2} = b$$

MoCs Considered in 421

Communication/ local computations	Shared memory	Message passing Synchronous	Asynchronous
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Combined Models

- Specification and Description Language (SDL)
  - FSM+asynchronous message passing
- StateCharts
  - FSM+shared memory
- Communicating Sequential Processes (CSP); ADA
  - von Neumann execution+synchronous message passing
- ….
- See also
  - Work by Edward A. Lee, UCB
  - Axel Jantsch: *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*, Morgan-Kaufman, 2004

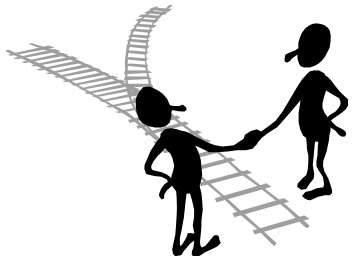
Ptolemy

- Ptolemy (UC Berkeley) is an environment for simulating multiple models of computation
  - http://ptolemy.berkeley.edu/

Facing reality

- No language meets all requirements

  $\Rightarrow$ Designers find the best compromise!



MoCs Considered in 421

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components		**Plain text, use cases | (Message) sequence charts**	
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Early Design Stages

- Capturing the requirements as text
  - Describe the system under design (SUD) in natural language
- Expectations for tools
  - Machine-readable
  - Version management
  - Dependency analysis
  - Example: DOORS® [Telelogic/IBM]
- How should requirements be captured?
  - Use cases
  - Message sequences

Use Cases

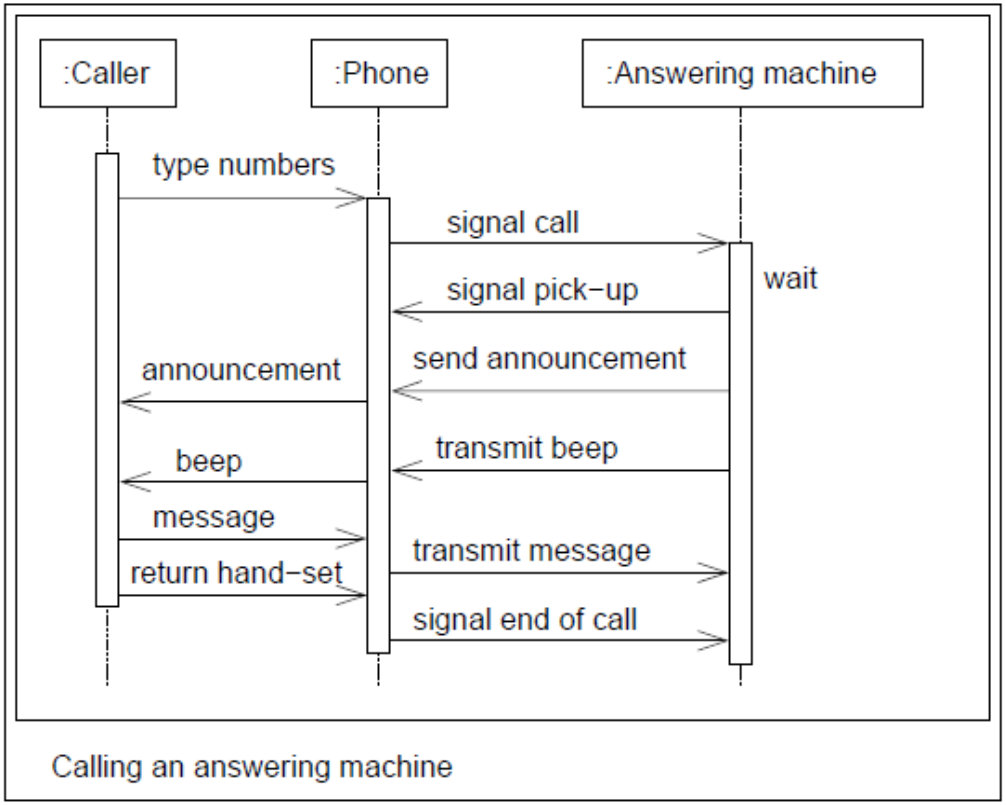
- Possible applications of the SUD
- Included in UML (Unified Modeling Language)
- Example: Answering machine

- Neither a precisely specified model of the computations nor a precisely specified model of the communication

(Message) Sequence Charts (MSC)

- Explicitly indicate exchange of information
- One dimension (usually vertical dimension) reflects time
- The other reflects distribution in space

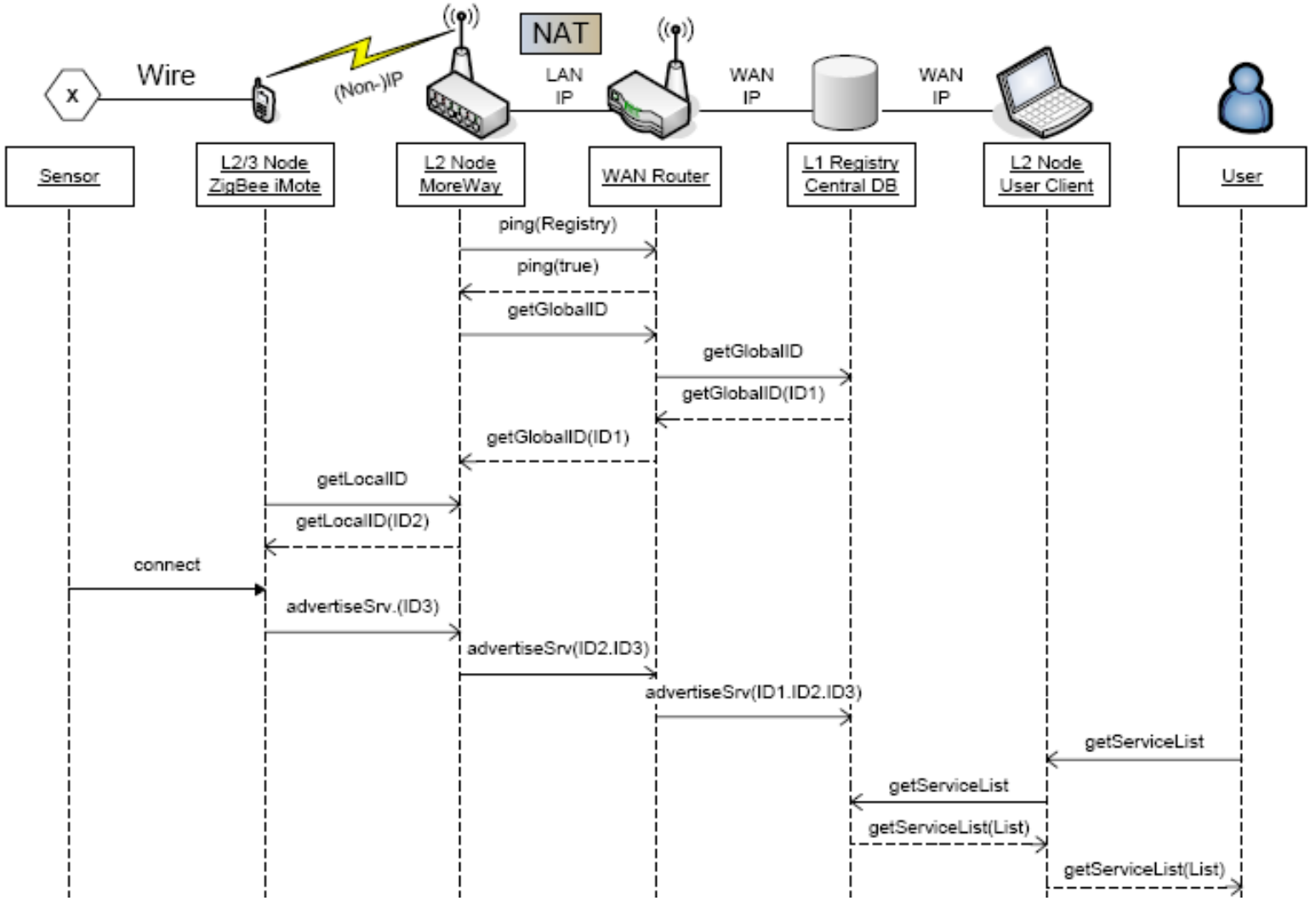
Example:



Calling an answering machine

- Included in UML
- Earlier called Message Sequence Charts, now mostly called Sequence Charts

Example: Sensor Services



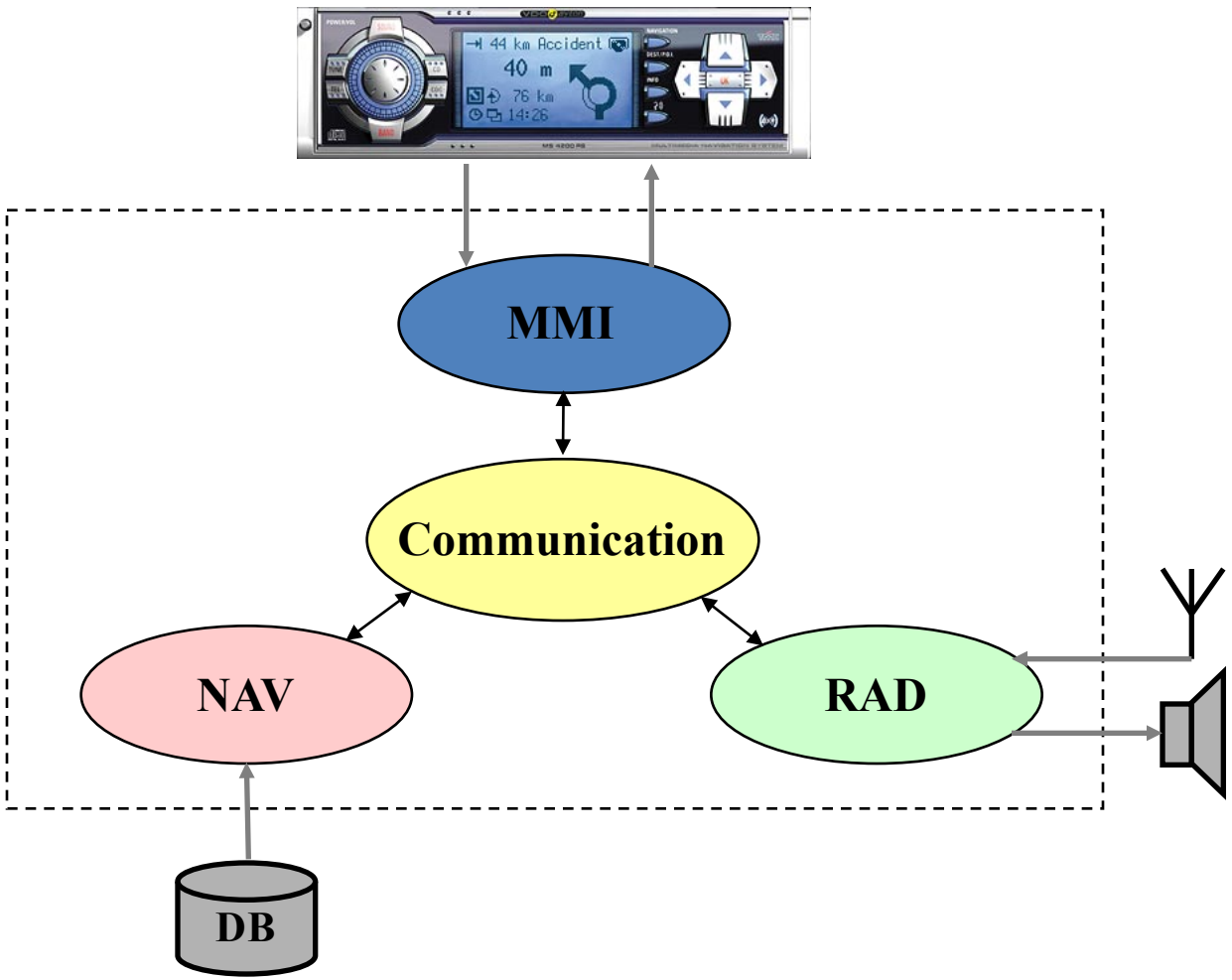
www.ist-more.org, deliverable 2.1

Example: In-Car Navigation System

- Car radio with navigation system
  - User interface should be responsive
  - Traffic messages (TMC) must be handled in a timely way
  - Several applications may execute concurrently

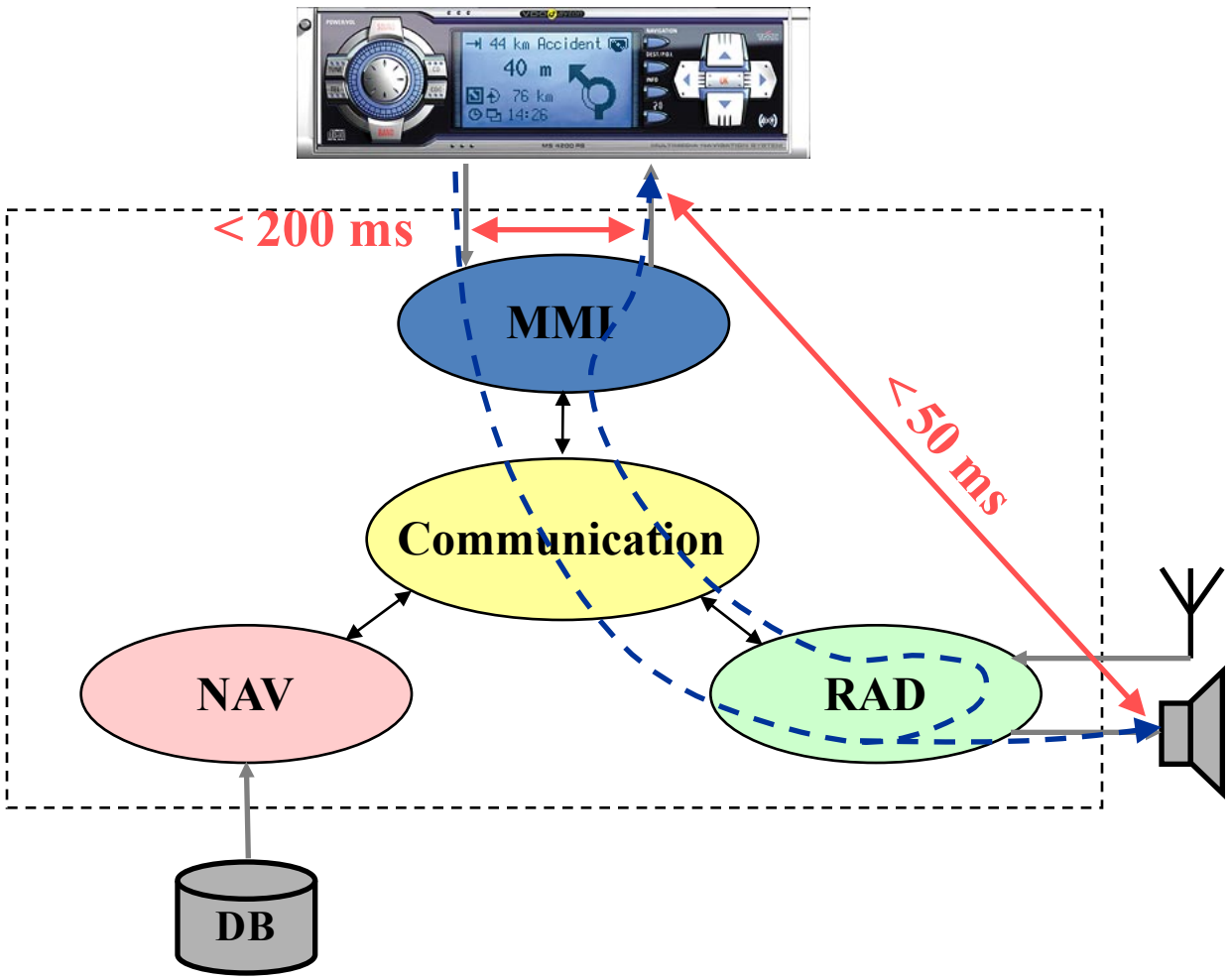
System Overview

© Thiele, ETHZ



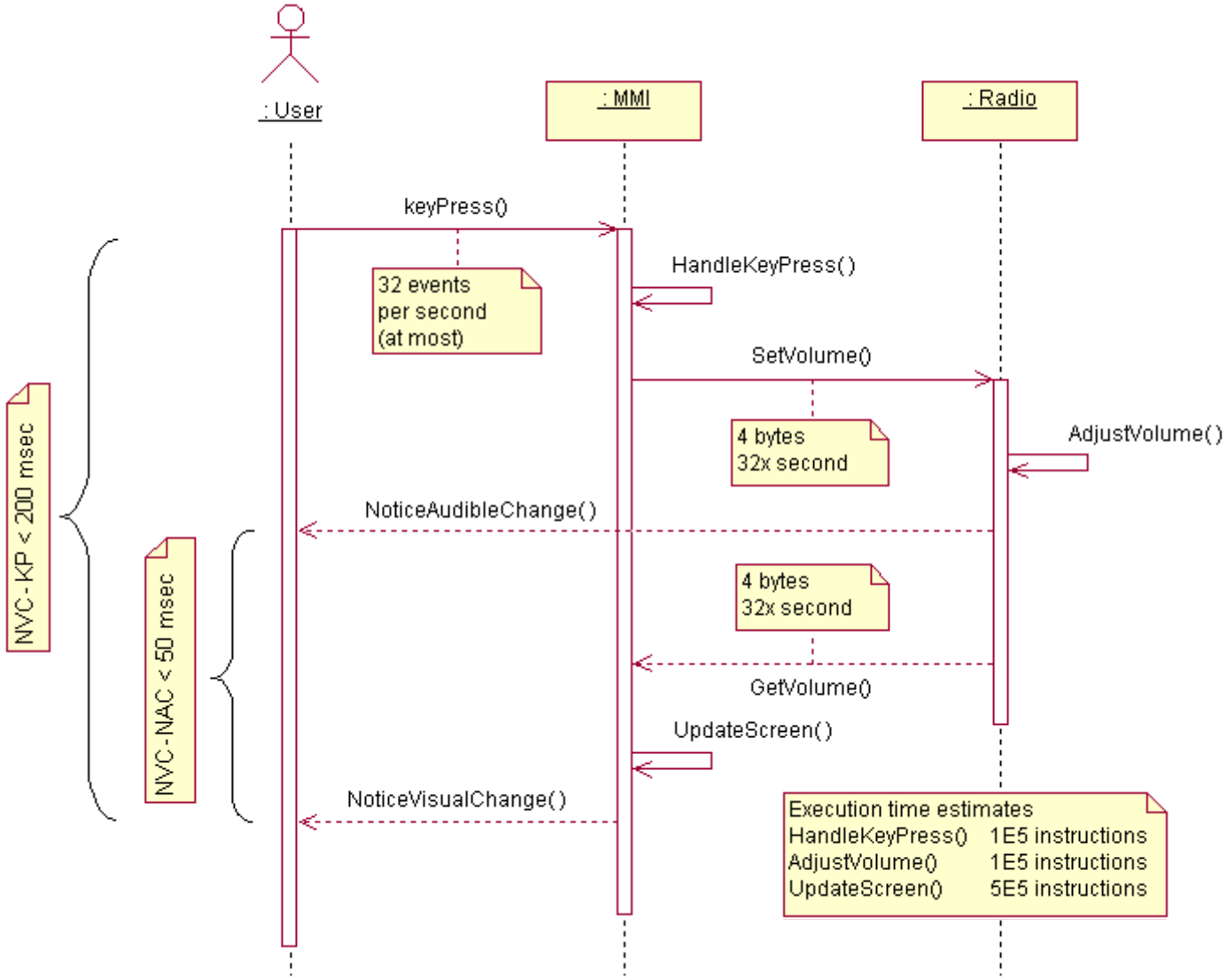
Use Case 1: Change Audio Volume

© Thiele, ETHZ



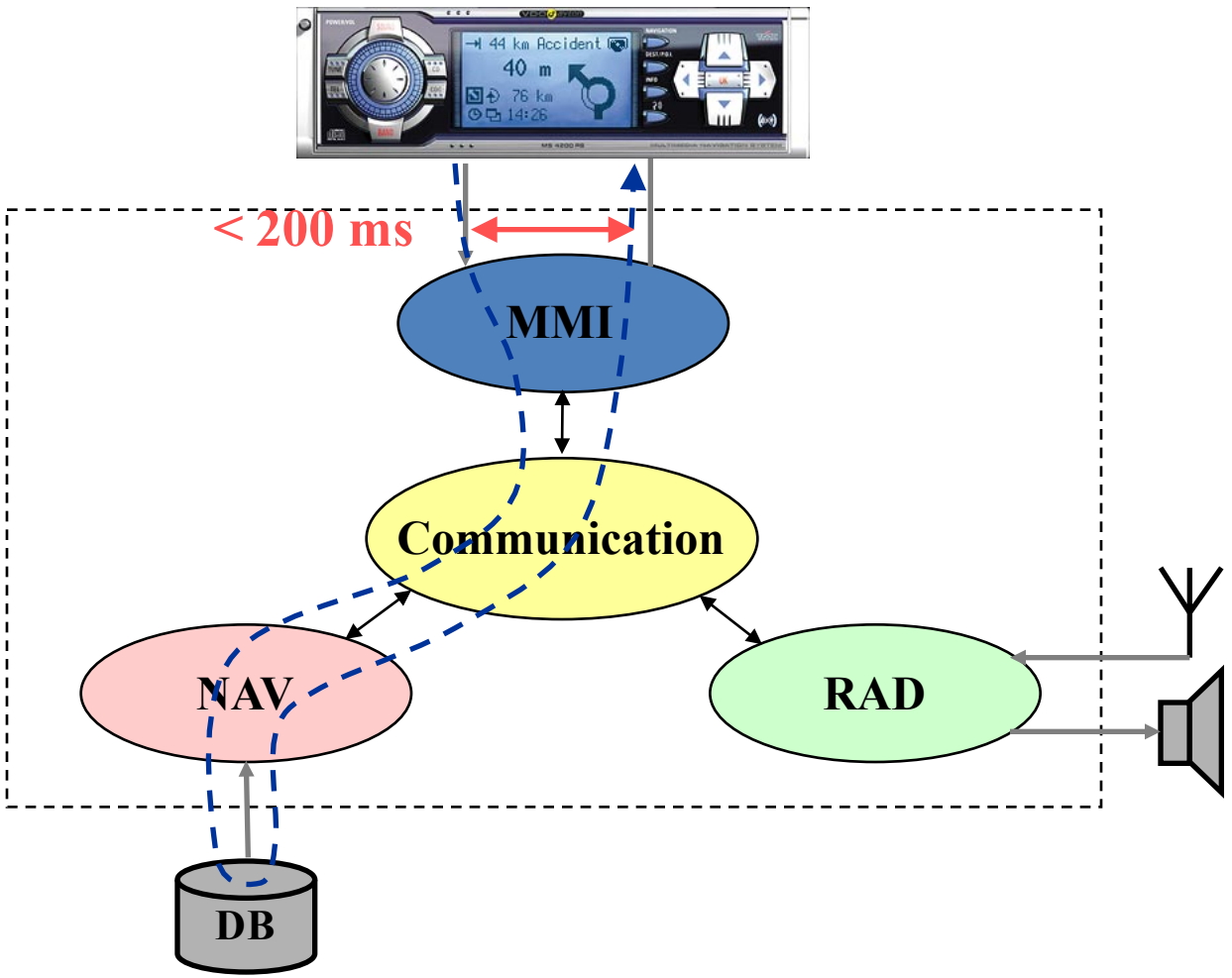
Use Case 1: Change Audio Volume

© Thiele, ETHZ



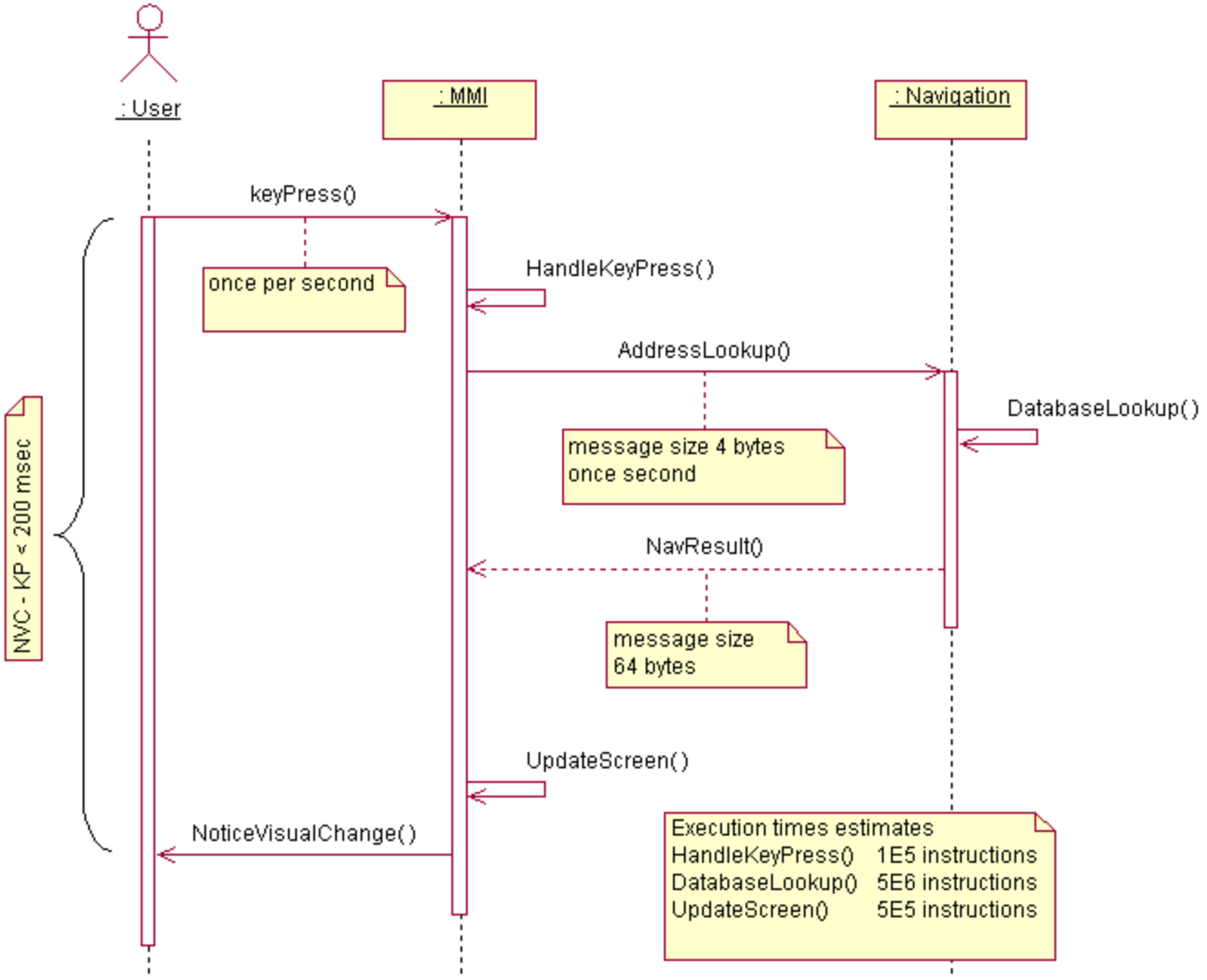
Use Case 2: Lookup Destination Address

© Thiele, ETHZ



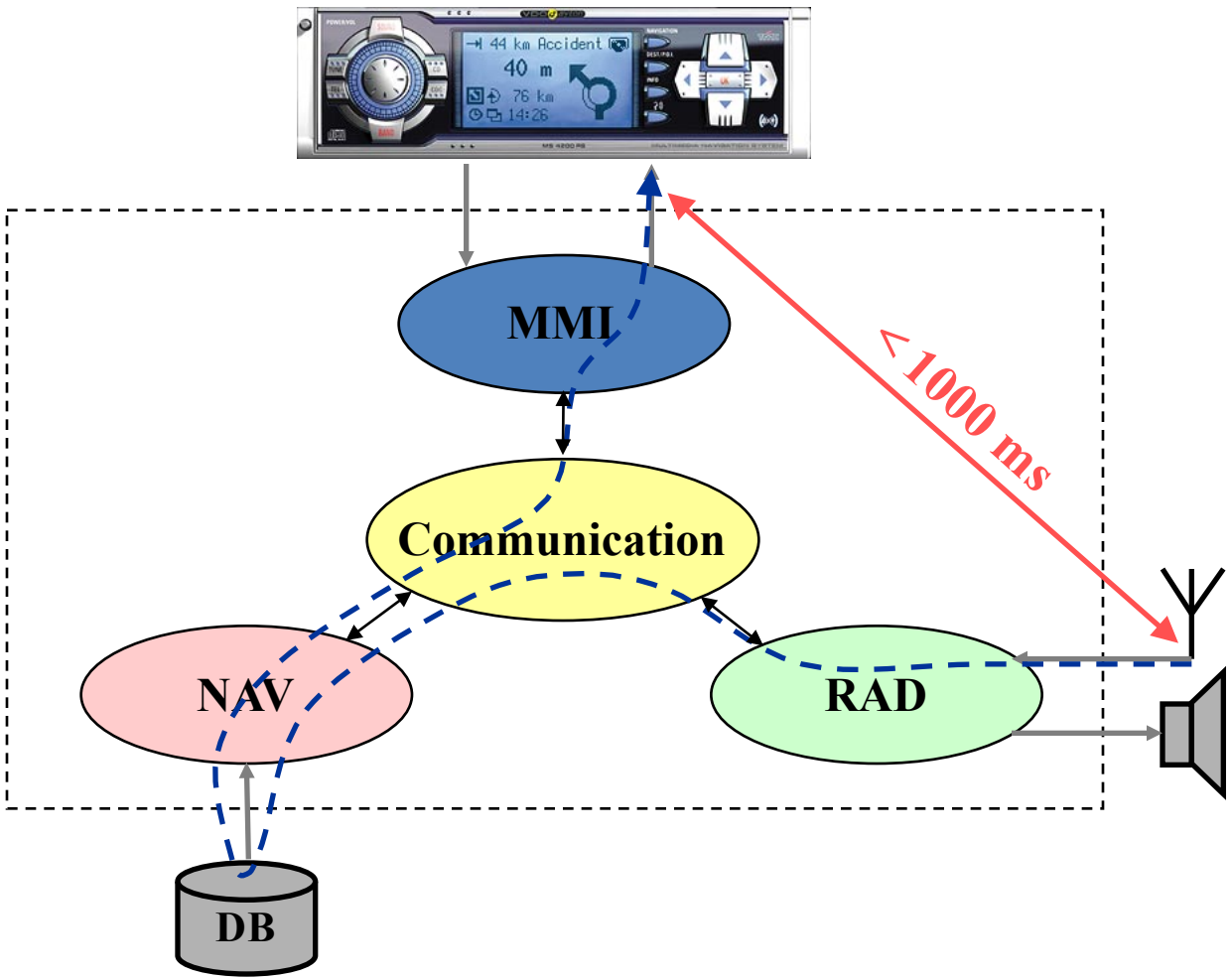
Use Case 2: Lookup Destination Address

© Thiele, ETHZ



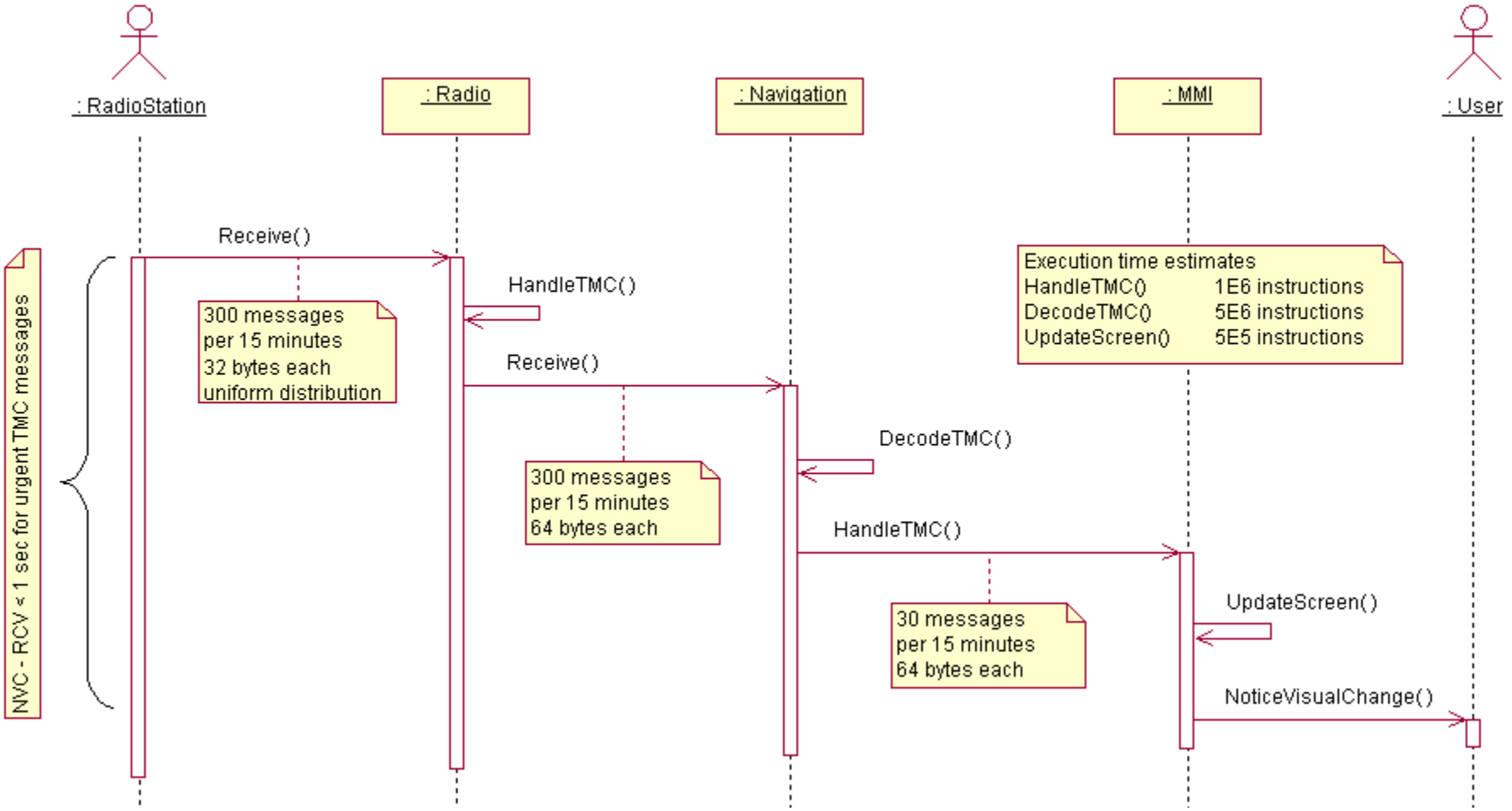
Use Case 3: Receive TMC Messages

© Thiele, ETHZ

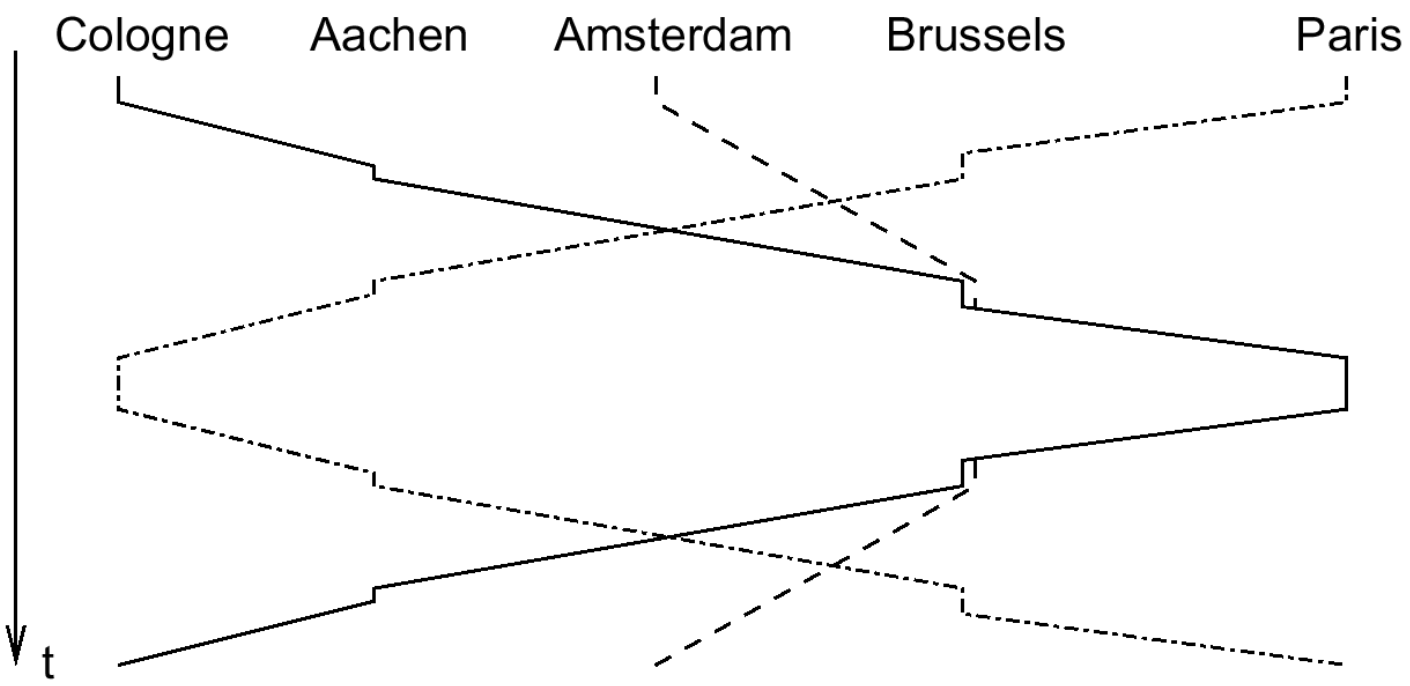


Use Case 3: Receive TMC Messages

© Thiele, ETHZ



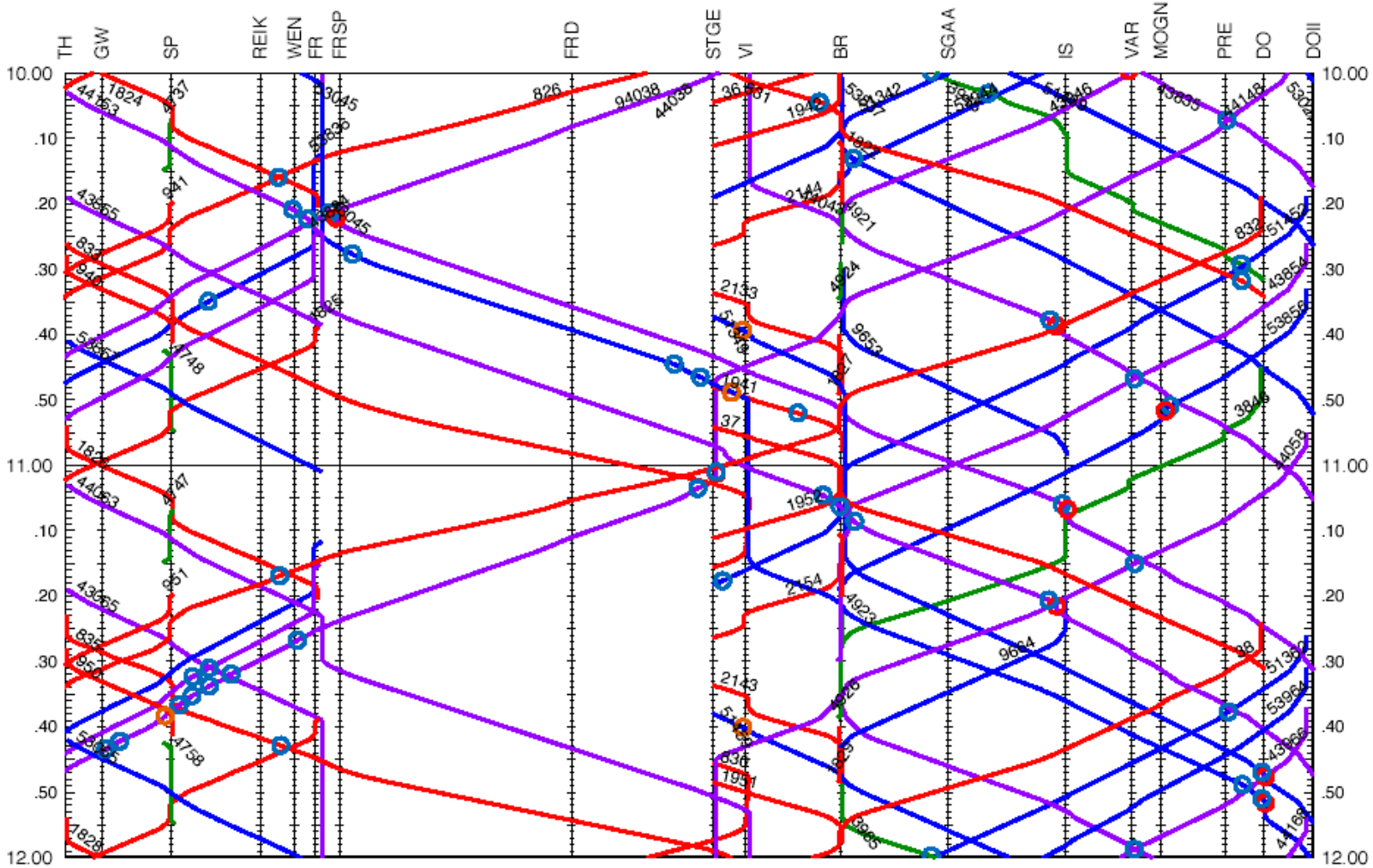
Concurrency in MSC



No distinction between accidental overlap and synchronization

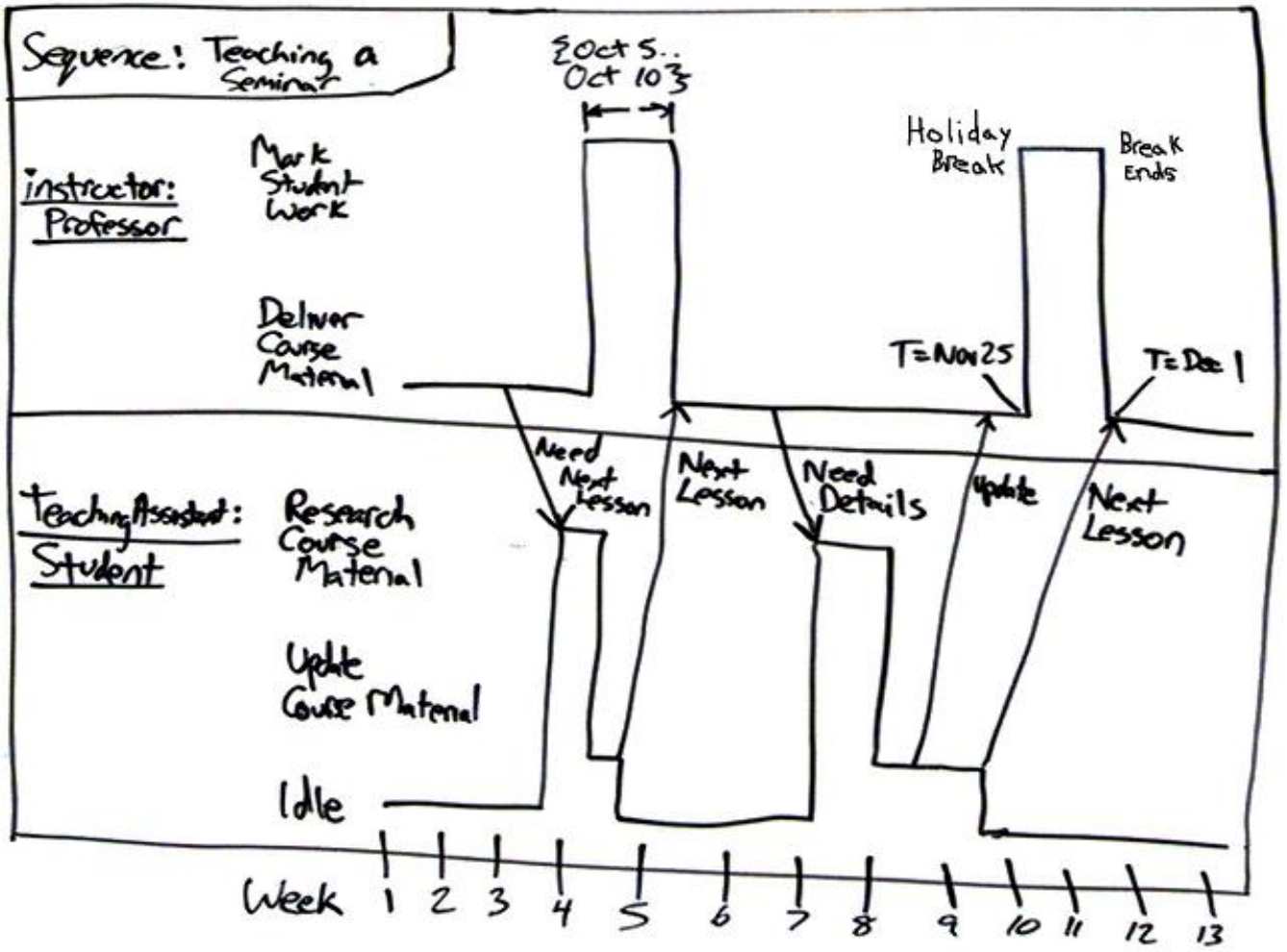
Time/distance Diagrams

- A special case of sequence charts



UML: Timing Diagrams

- Illustrate object state changes over time



© Scott Ambler,
Agile Modeling,
www.agilemodeling.com,
2003

(Message) Sequence Charts, Summarized

- **PROs**
  - Appropriate for visualizing schedules
  - Proven method for representing schedules in transportation
  - Standard defined: *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*, ITU-TS, Geneva, 1996
  - Semantics also defined: *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)—Annex* B: *Algebraic Semantics of Message Sequence Charts*, ITU-TS, Geneva
- **CONs**
  - Describes just one case!

Summary

- Introduction to requirements for specification languages
- Modeling Computation
  - Why not von Neumann?
  - Models for communication
  - Models of components
- Task graphs
- Supporting early design phases
  - Textual descriptions
  - Use cases or scenarios
  - (Message) sequence charts

Next Time

- Communicating Finite State Machines (CFSMs)
  - Chapter 2.4