



UMLEmb:

UML for Embedded Systems

II. Modeling in SysML

Ludovic Apvrille,
ludovic.apvrille@telecom-paris.fr

LabSoC, Sophia-Antipolis, France



Outline

Case Study

Method

Case Study: a Pressure Controlling System

A "client" expects you to deliver the software of the following system:

Specification (from the client)

- A pressure controller informs the crew of a cabin with an alarm when the pressure exceeds 20 bars in the cabin
- The alarm duration equals 60 seconds.
- Two types of controllers. "Type 2" keeps track of the measured values.



Pressure Controller: Assumptions

Modeling assumptions linked to the system

- The controller set up and shutdown procedures are not modeled
- The controller maintenance is not modeled
- Versioning
 - The "keep track of measured value" option is not modeled in the first version of the design

Modeling assumptions linked to the system environment

- The pressure sensor never fails
- The alarm never fails
- The controller never faces power cut



Outline

Case Study

Method

Requirements

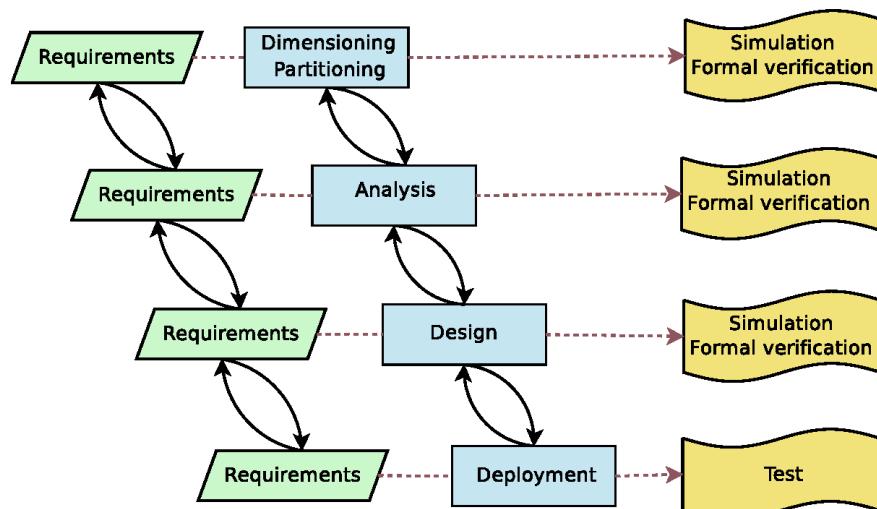
(Partitioning)

Analysis

Design



Overview of the V Cycle



Outline

Case Study

Method

Requirements

(Partitioning)

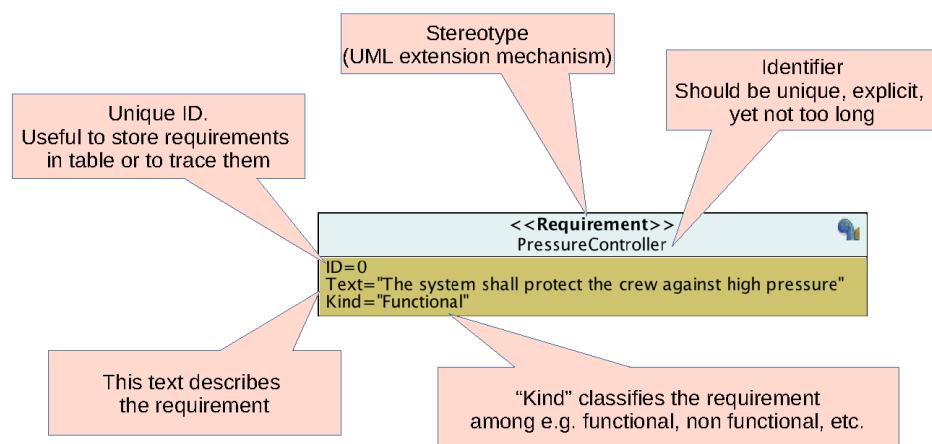
Analysis

Design



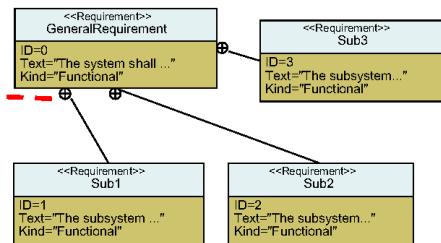
Requirement Node

- A requirement node identifies a requirement by:
 - A unique identifier (so as to achieve traceability)
 - A description in plain text
 - A type (functional, non functional, performance, security, ...).

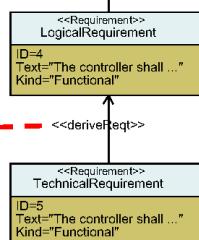


Relations Between Requirement Nodes

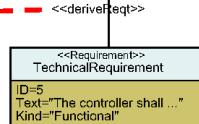
Containment relation -
Splits up a compounded requirement into elementary ones



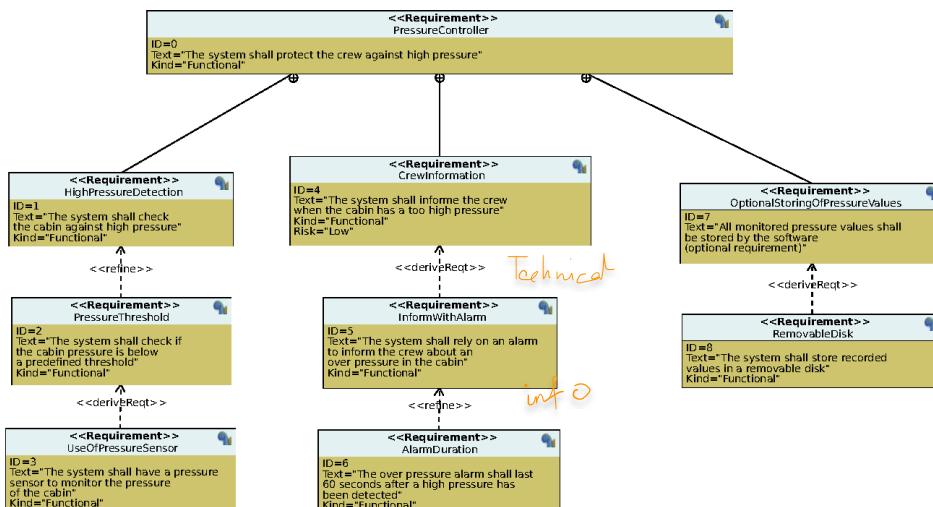
Refinement -
Relates two requirements of different abstraction levels



Derivation -
Builds a new requirement from the reuse of other requirements

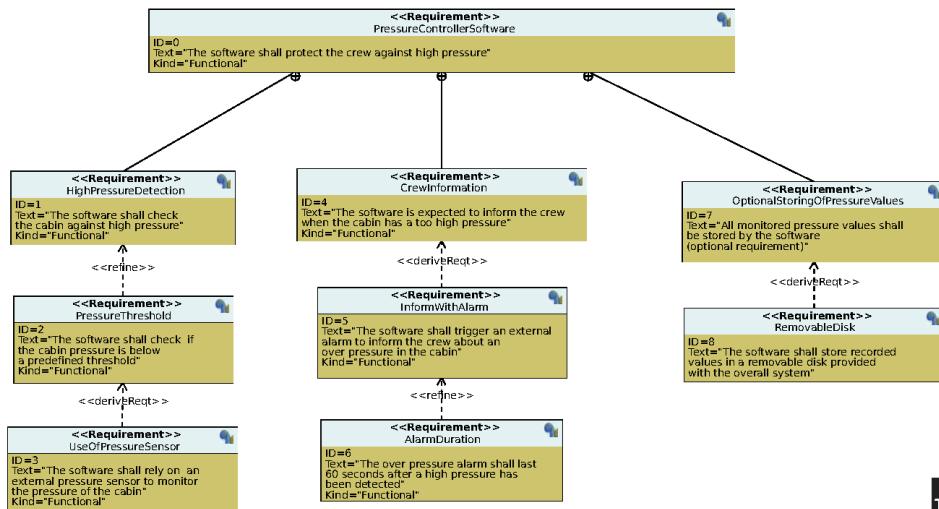


Requirement Diagram - Pressure Controller - System View



Summarize RUP

Requirement Diagram - Pressure Controller - Software View



Outline

Case Study

Method

Requirements

(Partitioning)

Analysis

Design

Complex Embedded Systems

- Complex Embedded System = set of SW and HW components intended to perform a predefined set of functions for a given market
- Constraints
 - Right market window
 - Performance and costs



Design Challenges



Complexity

- Very high software complexity
- Very high hardware complexity

Problem

How to decide whether a function should be implemented in SW or in HW, or both?



Solution

Design Space Exploration!
 (a.k.a. "Partitioning")



Design Space Exploration

Design Space Exploration

- Analyzing various functionally equivalent implementation alternatives
- → Find an optimal solution

Important key design parameters

- Speed
- Power Consumption
- Silicon area
- Generation of heat
- Development effort
- ...



Level of Abstraction

Problematic

- Designers struggle with the complexity of integrated circuits (e.g. System-on-chip)
- Cost of late re-engineering
 - Right decisions should be taken as soon as possible ...
 - And quickly (time to market issue), so simulations must be fast

→ System Level Design Space Exploration

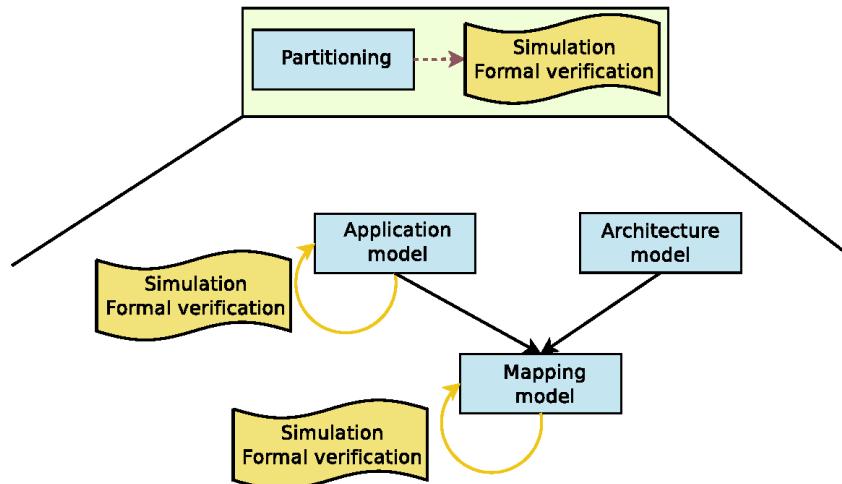
- Reusable models, fast simulations / formal analysis, prototyping can start without all functions to be implemented

But: high-level models must be closely defined so as to take the right decisions (as usual ...).

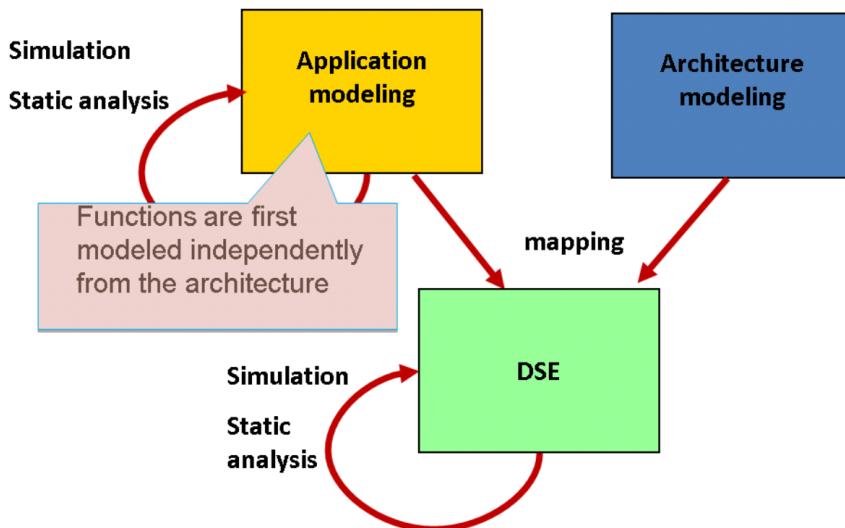


Partitioning with the Y-Methododology

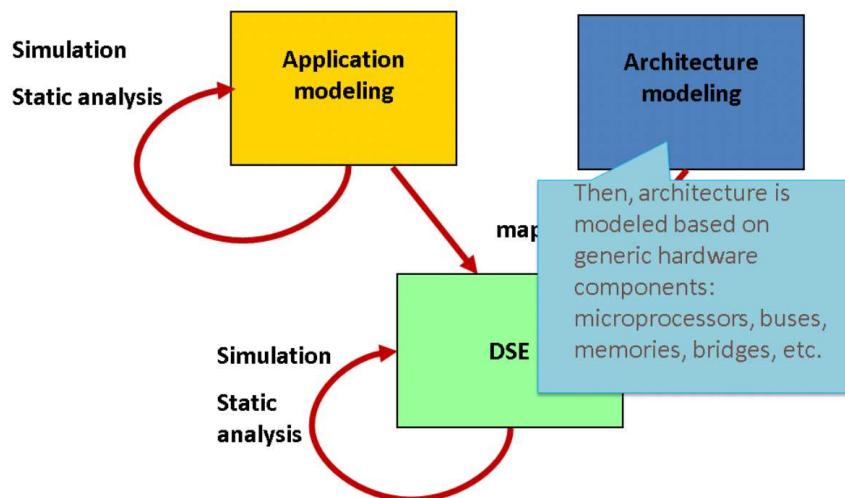
- Example: the DIPLODOCUS methodology



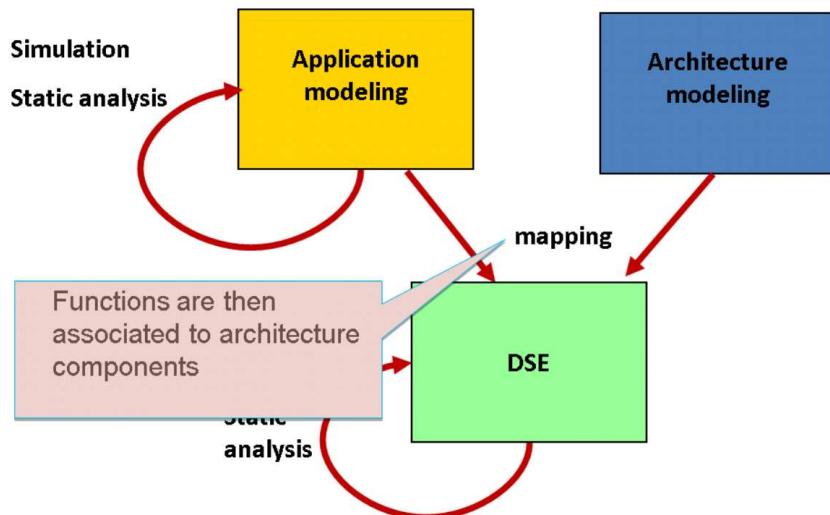
Application Modeling



Architecture Modeling



Mapping



Outline

Case Study

Method

Requirements

(Partitioning)

Analysis

Design



System Analysis

Analysis = Understanding what a client wants

- So, it does not mean "creating a system", but rather "understanding the main functionalities" of the system to be designed
- Can be performed before or after the partitioning stage

Analysis method

1. System boundary and main functions → *Use Case Diagram*
2. Relations between main functions → *Activity Diagram*
3. Communications between main system entities and actors → *Sequence Diagram*



Use Case Diagram: Method

■ Shows what the system does and who uses it

1. Define the boundary of the system

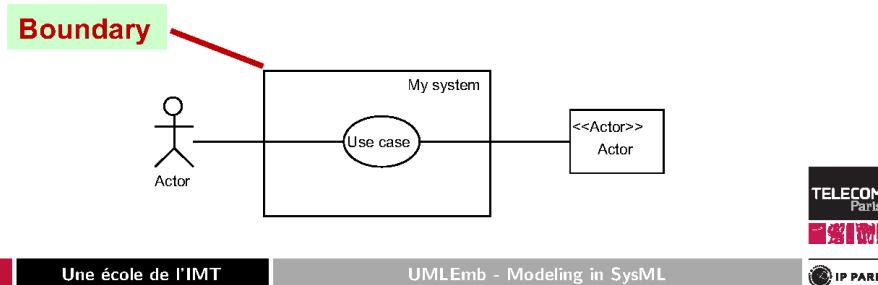
- Inside of the rectangle → What you promise to design
- Outside of the rectangle → System environment (= Actors)
- This is not part of what you will have to design

2. Name the system

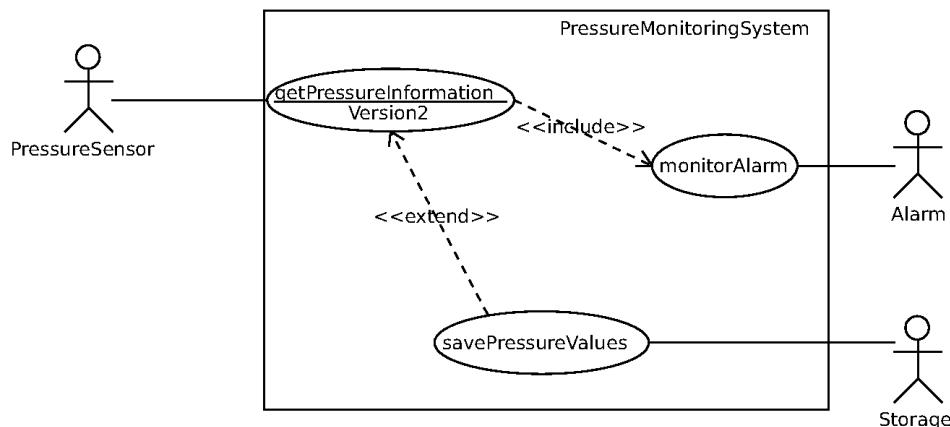
3. Identify the services to be offered by the system

- Only services interacting with actors

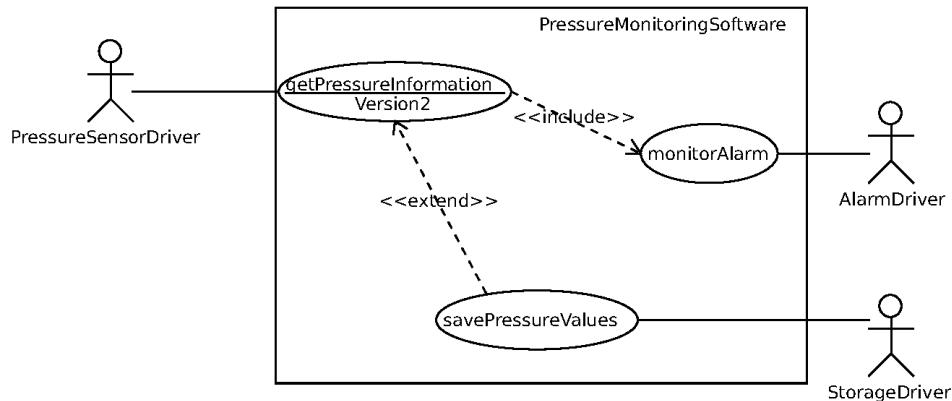
4. Draw interactions between functions and actors



Use Case Diagram - Pressure Controller - System View

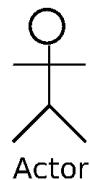


Use Case Diagram - Pressure Controller - Software View



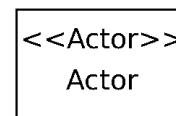
Actors

■ Syntax 1: Stickman



Actor

■ Syntax 2: <<Actor>>

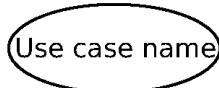


Method

- An actor identifier is a substantive
- An actor must interact with the system

Use Case

- **Syntax:** ellipse with exactly one use case



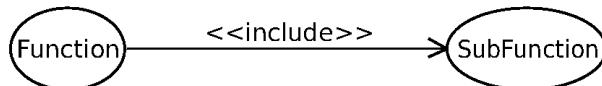
Method

- A use case is described by a verb
 - The verb should describe **the point of view of the system**, not the point of view of the actors
- A use case diagram must **NOT** describe a step-by-step algorithm
 - A use case describes a high-level service/function, not an elementary action of the system

Use Case to Use Case Relations

■ Inclusion

- A function mandatorily includes another function



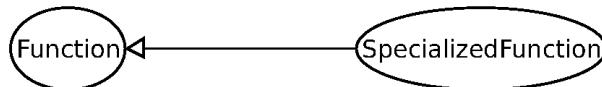
■ Extension

- A function optionally includes another function

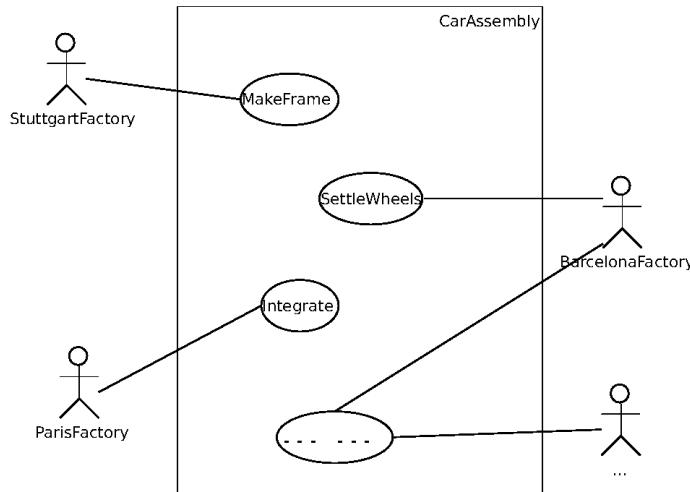


■ Inheritance

- A "child" function specializes a "parent" function

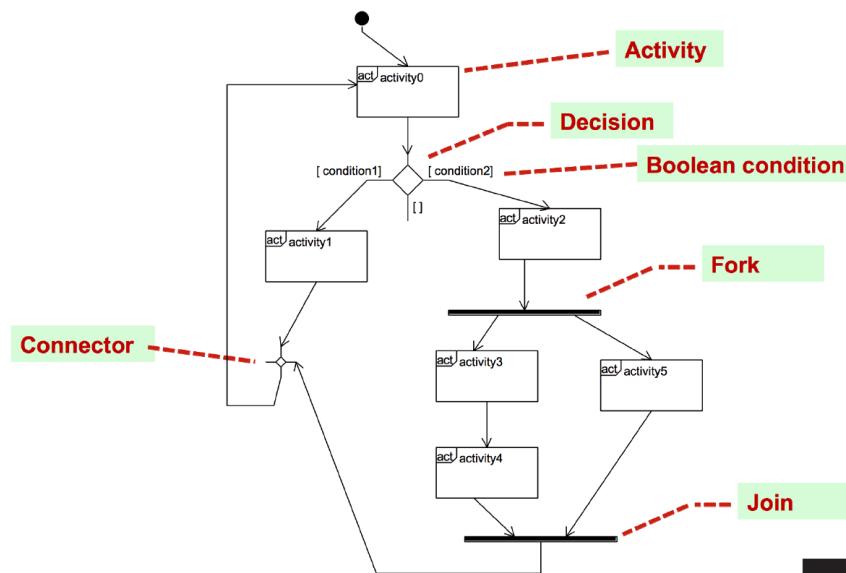


Location-Driven Use Case Diagram

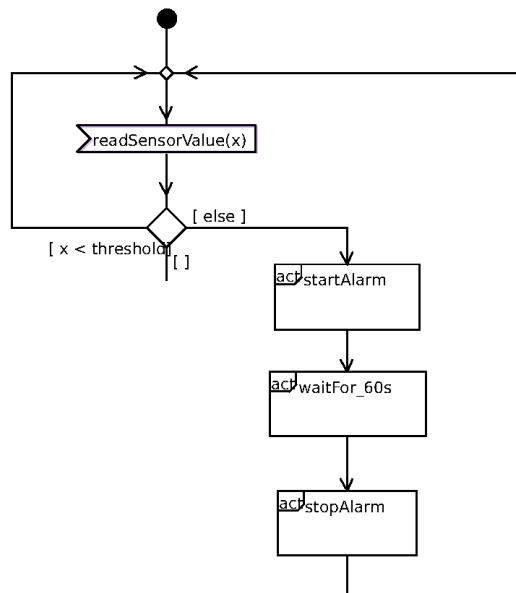


Activity Diagram - Syntax

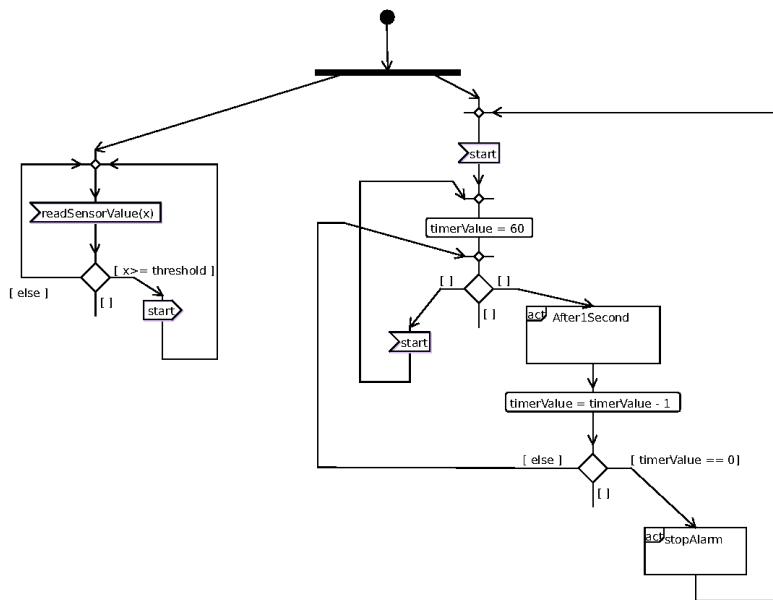
Shows functional flows in the form of succession of actions



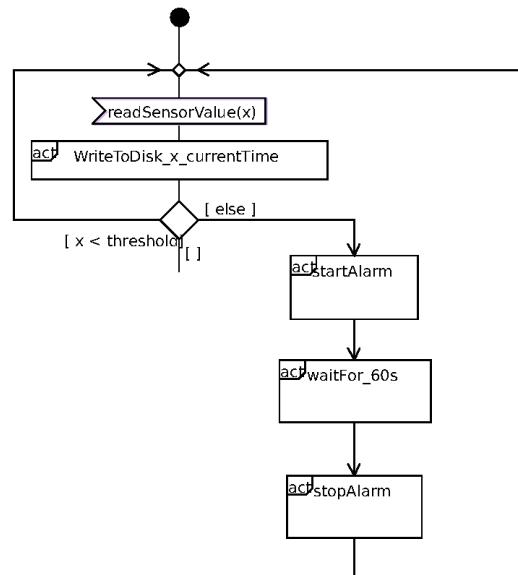
Activity Diagram - Pressure Controller



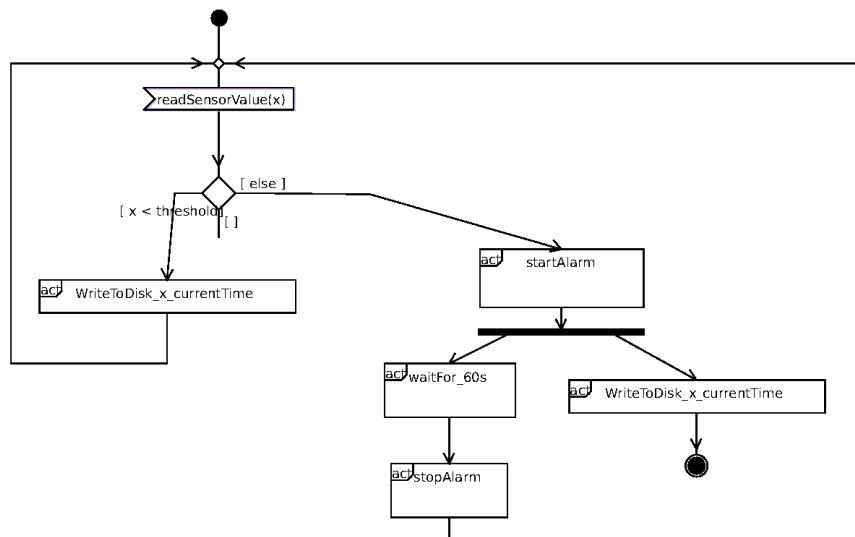
Activity Diagram - Pressure Controller



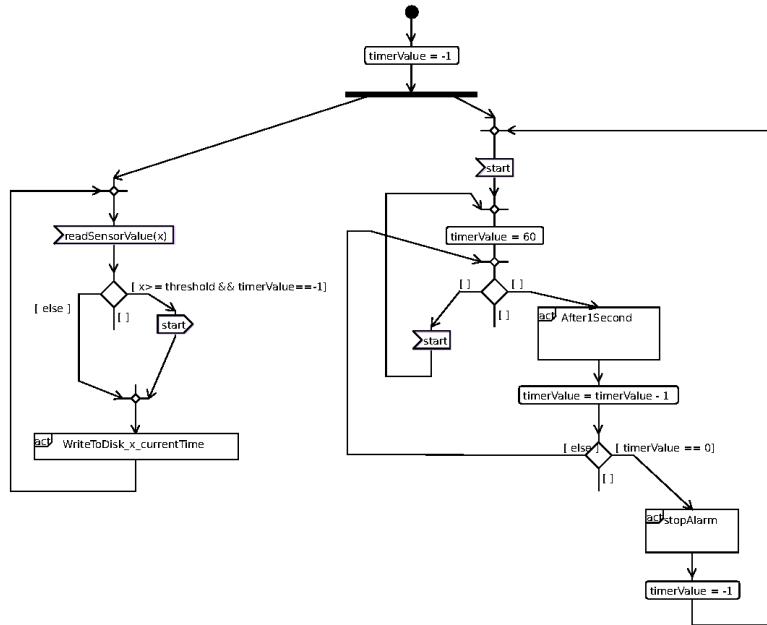
Activity Diagram - Pressure Controller



Activity Diagram - Pressure Controller

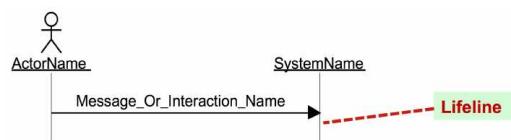


Activity Diagram - Pressure Controller

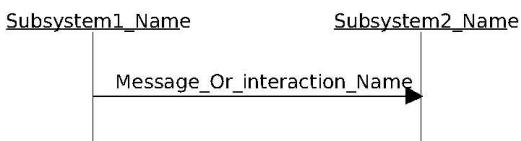


Sequence Diagram

- An actor interacting with a system

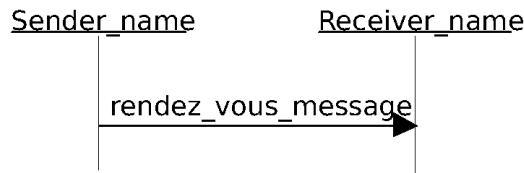


- Two interacting "parts" of the system

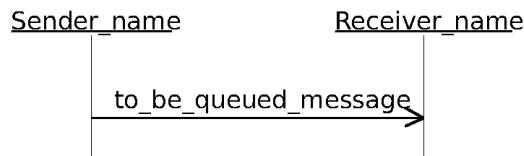


Sequence Diagram - Messages

- **Synchronous communication** (black arrow)



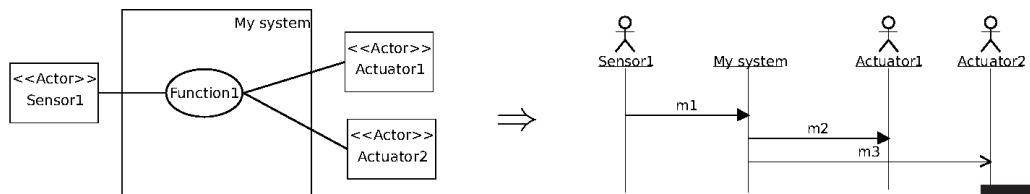
- **Asynchronous communication** (regular arrow)



Using Sequence Diagrams

Method

- A sequence diagram depicts one possible execution run, **NOT** the entire behavior of the system
- **NO message between actors**
- All actors must be defined in the use case diagram
 - WARNING: Coherence between diagrams

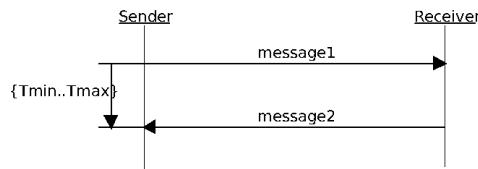


Sequence Diagram - Time (1/2)

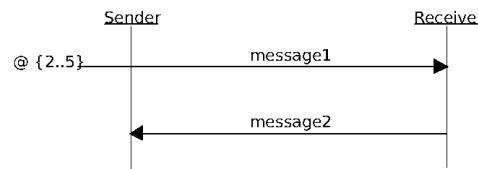
Semantics

- One global clock (applies to the entire system)
- Time uniformly progresses (lifelines are read top-down)
- Causal ordering of events on lifelines
 - Time information must be explicitly modeled

■ Relative dates

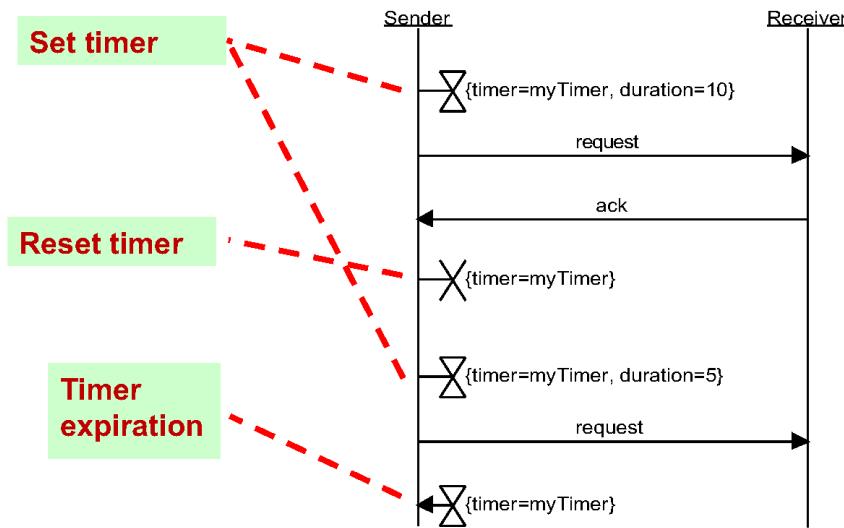


■ Absolute date



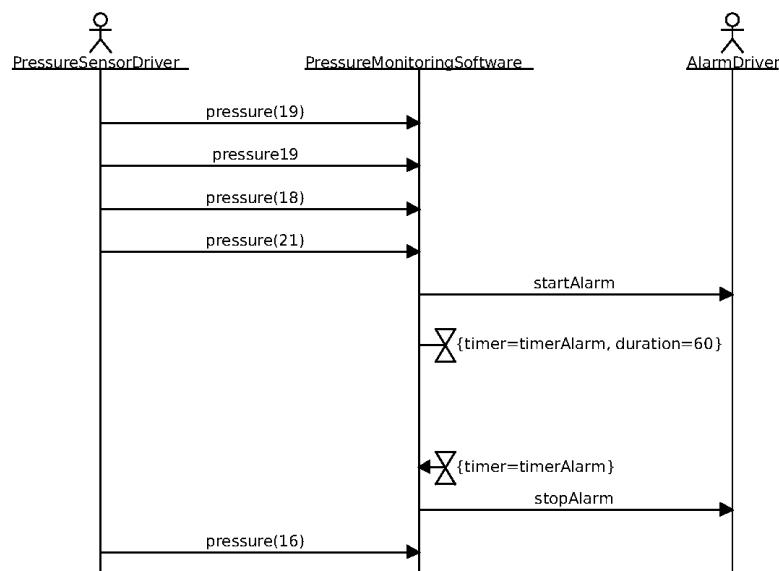
Sequence Diagram - Time (2/2)

■ Timers



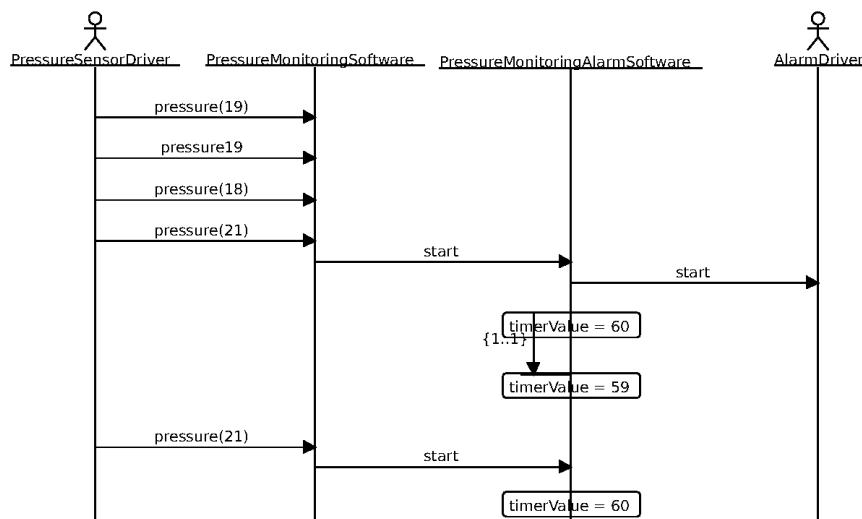
Sequence Diagram - Pressure Controller

Nominal trace



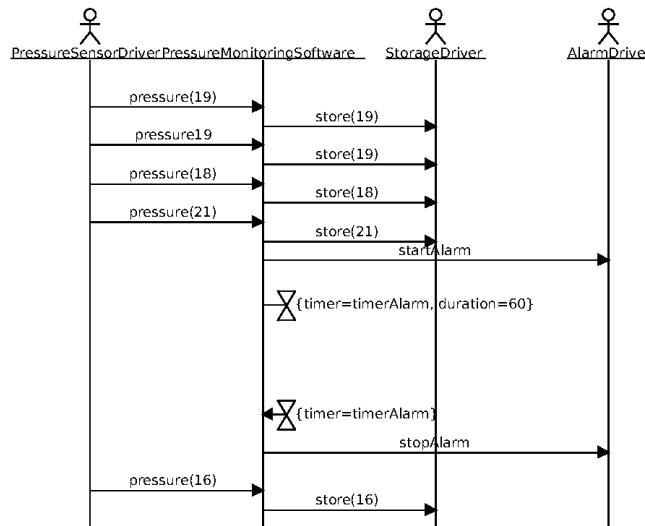
Sequence Diagram - Pressure Controller

Advanced trace



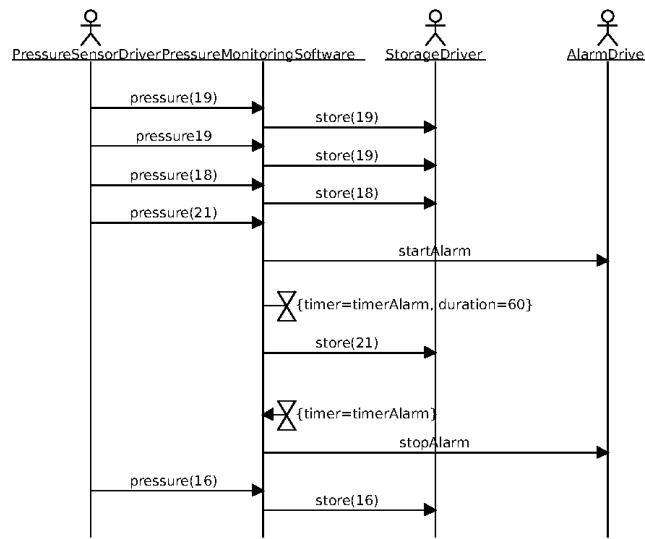
Sequence Diagram - Pressure Controller

Nominal trace - version 2 (save before alarm)



Sequence Diagram - Pressure Controller

Nominal trace - Version 2 (save after alarm)



Outline

Case Study

Method

Requirements

(Partitioning)

Analysis

Design



System Design

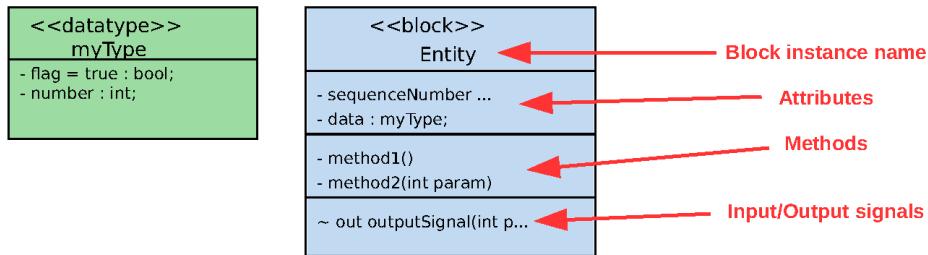
Design = Making what a client wants

So, it means "inventing a system", "creating a system" that complies with the client requirements.

- System architecture → *Block Definition Diagram and Internal Block Diagram*
 - In AVATAR, they are merged in one diagram that contains:
 - The definition of blocks
 - The interconnection of these blocks
- Behaviour of the system → *State Machine Diagram*
 - One state machine diagram per block



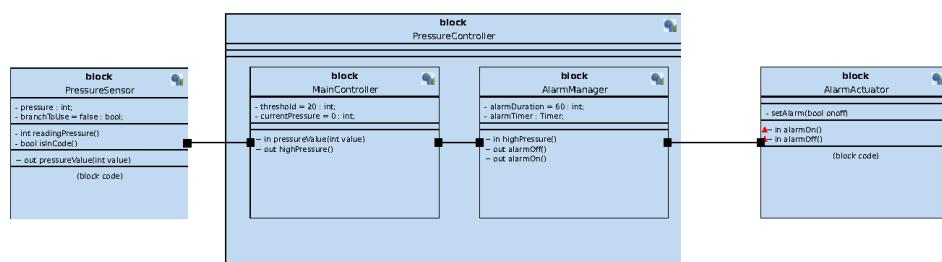
(Instance) Block Diagram: Syntax of Blocks



Note

- Upper-case and lower case characters
- Attributes are *private* elements
- Signals have the *package* access right
 - Package = a block + its sub-blocks

(Instance) Block Diagram: Connecting Blocks

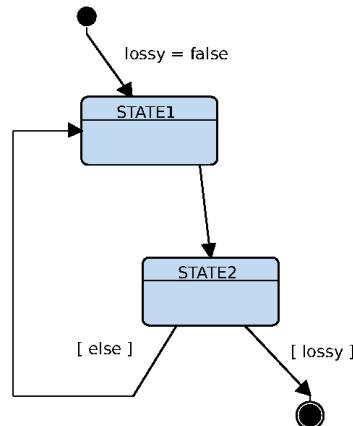


- Ports are connected to allow the state machines of blocks to exchange signals
- A block instance may nest one or several block instances

State Machine - States and Transitions

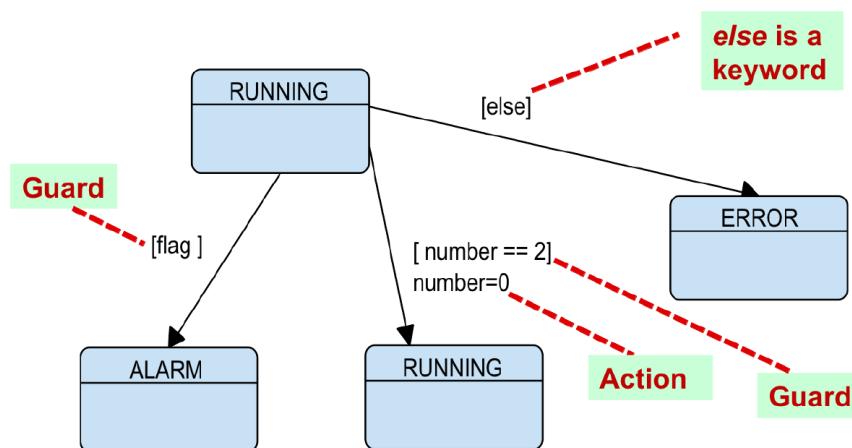
Note

- No parallelism
- States can have several output transitions



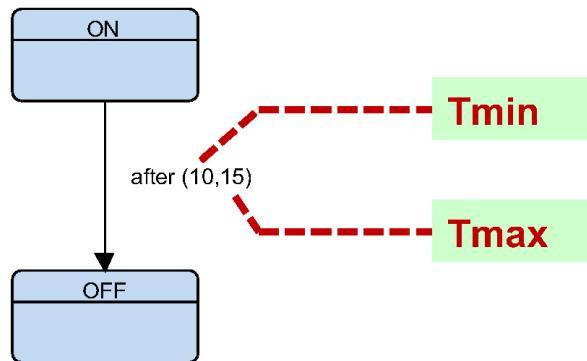
State Machines - Guards

- A transition guard contains a *boolean expression* built upon boolean operators and attributes



State Machines - Time Intervals

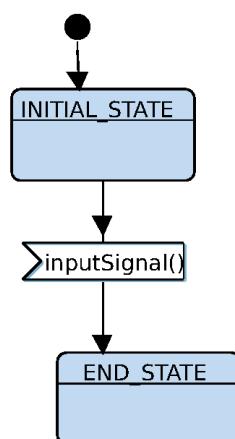
- after clause with a $[T_{min}, T_{max}]$ interval



- A transition with no *after* clause has de facto an **after(0,0)** clause, which means the transition may be fired "immediately"

State Machine - Inputs (1/3)

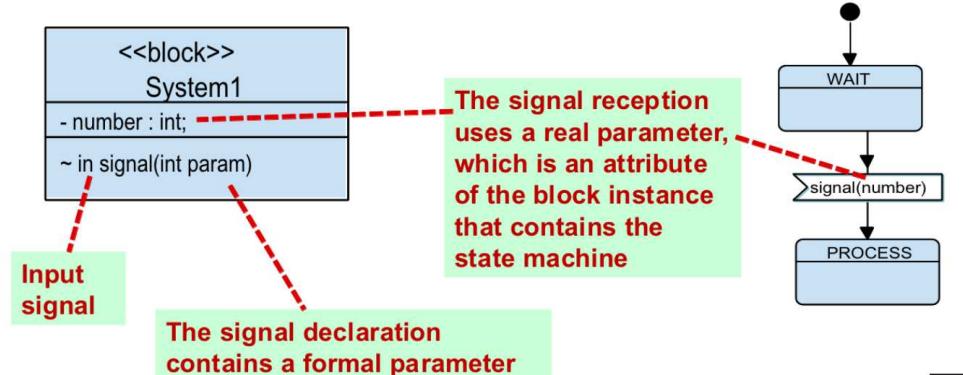
- A signal reception is a **transition trigger**



- The transition between INITIAL_STATE and END_STATE is triggered by a signal reception
- **Asynchronous communication**
 - FIFO-based
 - The transition is fired if $\text{size}(FIFO, inputSignal) > 0$
- **Synchronous communication**
 - The transition is fired whenever a rendezvous is possible
- Signals can convey parameters

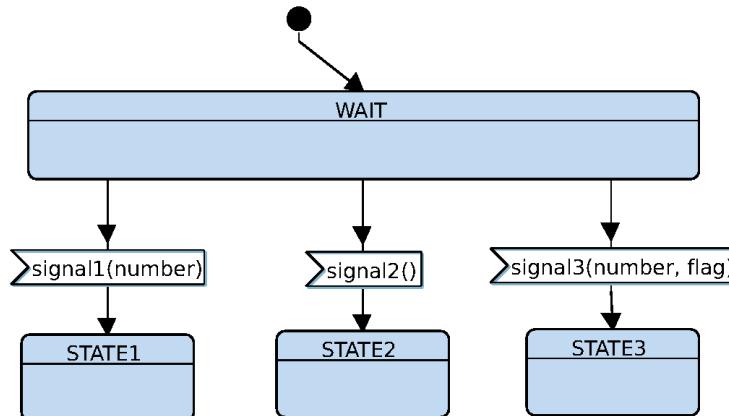
State Machine - Inputs (2/3)

- Signal parameters, if any, are stored in attributes of the block instance that receives the signal



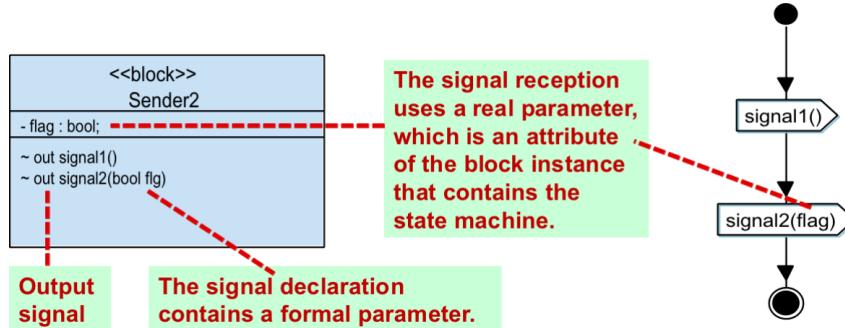
State Machine - Inputs (3/3)

- From the same state it is possible to wait for several signals
 - Asynchronous communication: the first signal in the input queues triggers the transition
 - Synchronous communication: The first ready-to-execute rendezvous triggers the transition



State Machine - Outputs

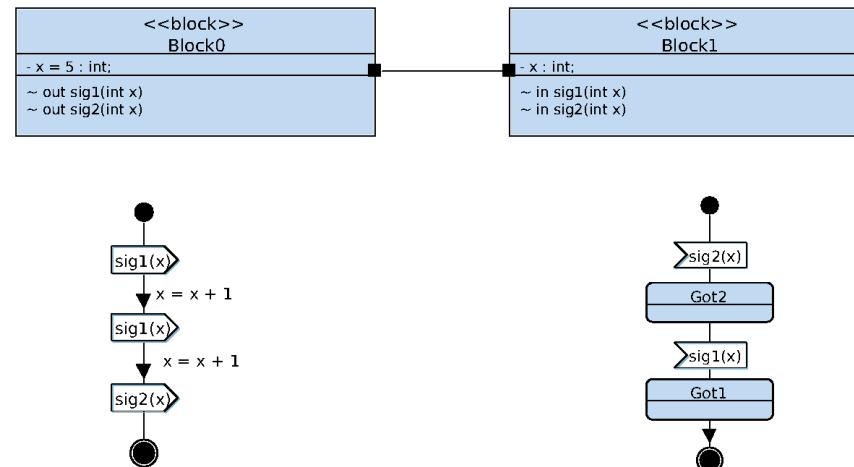
- A block instance can send signals with several parameters
 - Constant values may not be used as real parameters → use attributes instead



- A block instance cannot send two or several signals in parallel but it can send two or more signals in sequence

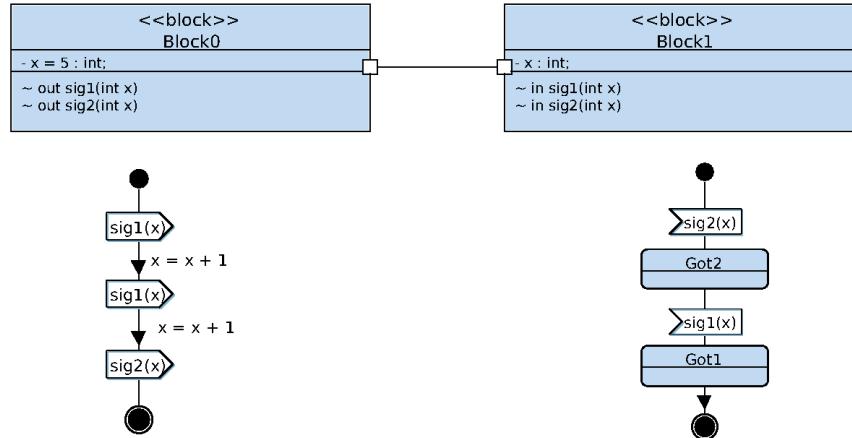
Synchronous Communications

- Sender and receiver synchronizes on the same signal
- Data exchange from the writer to the reader



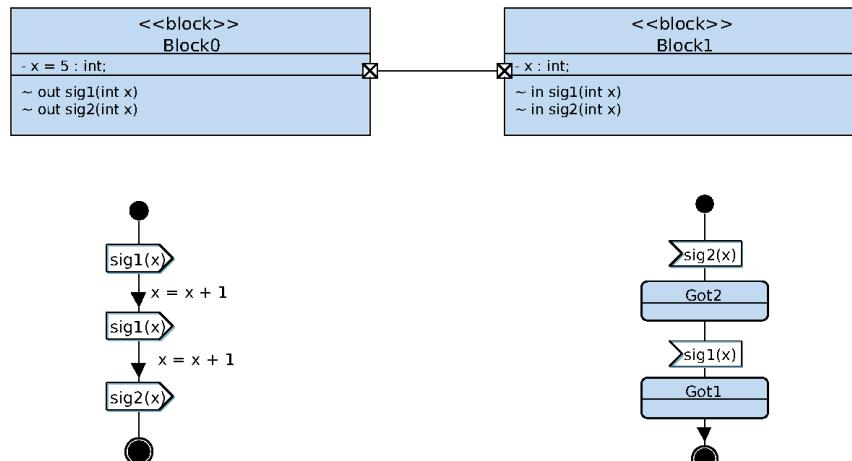
Non-Blocking Asynchronous Communications

- One FIFO per signal association
- Writing is **NOT** blocked when the FIFO is full
 - Bucket approach when FIFO is full: new messages are dropped
- Example: we assume a FIFO of size 1



Blocking Asynchronous Communications

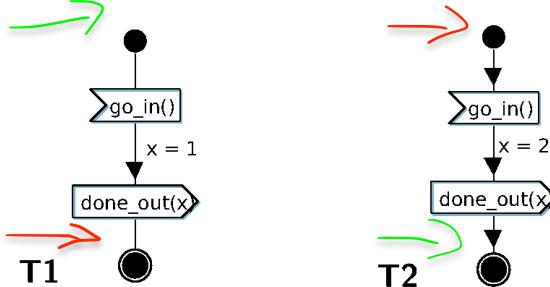
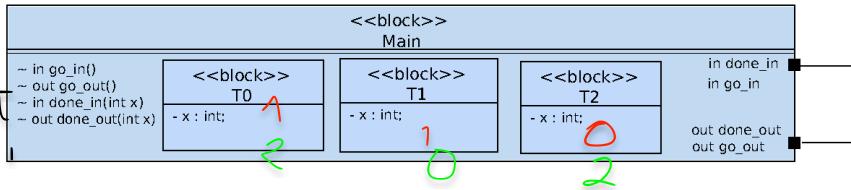
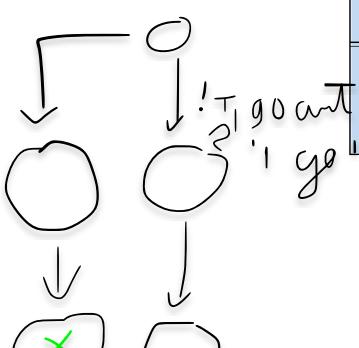
- One FIFO per signal association
- Writing is blocked when the FIFO is full
- Example: we assume a FIFO of size 1



State Machine - Advanced I/O

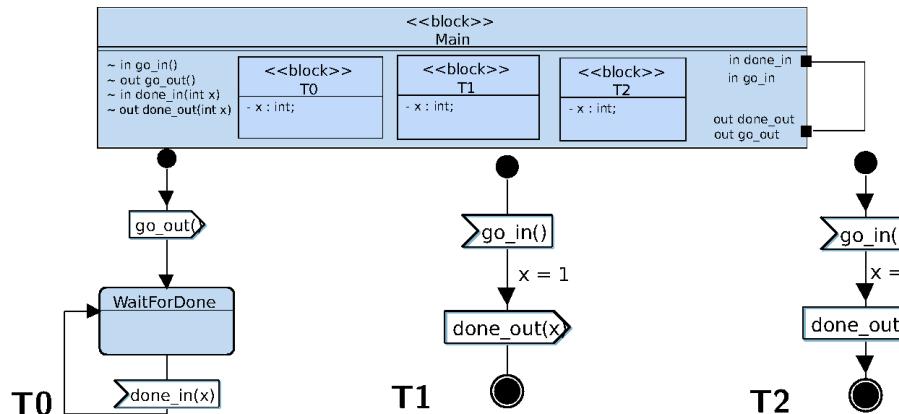
- Signals declared by a block may be used by its sub-blocks

R. 6



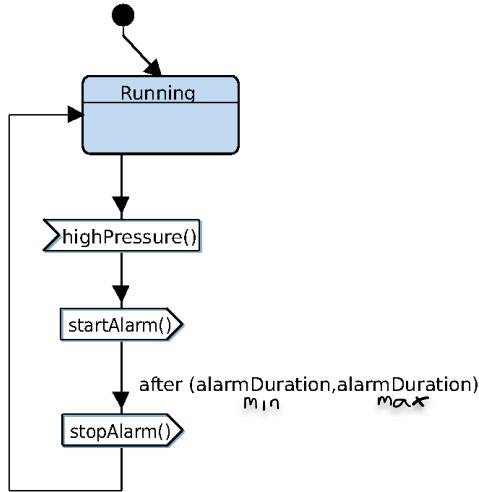
Broadcast Channel

- All blocks ready to receive a signal sent over a broadcast channel receive it
- So, what happens if the channel below is now set to *broadcast*?



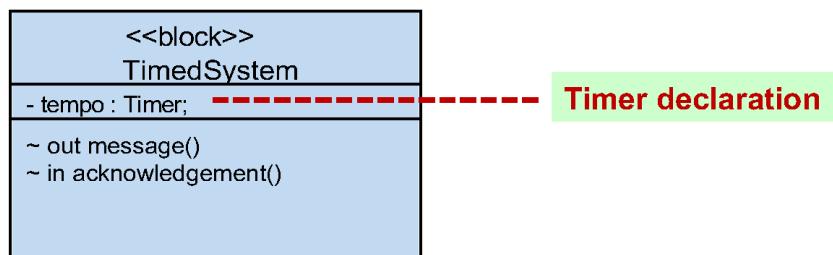
State Machine Diagram - Pressure Controller

- Shows the inner functioning of the *Controller* block instance



State Machines - Timers (1/3)

- A timer must be declared as an attribute of the block instance which uses it
 - Unlike attribute declarations, a timer declaration cannot contain an initial value
 - Use the *set* operator to initialize the duration of a timer
 - The signal issued by the timer at expiration time does not need to be declared



State Machines - Timers (2/3)

Set

- The "set" operation starts a timer with a value given as parameter
- The timer is based on a global system clock

Reset

Prevents a previously set timer to send an expiration signal

Expiration

- A timer "timer1" sends a signal named "timer1" to the block instance it belongs to
- ⇒ A timer expiration is handled as a signal reception

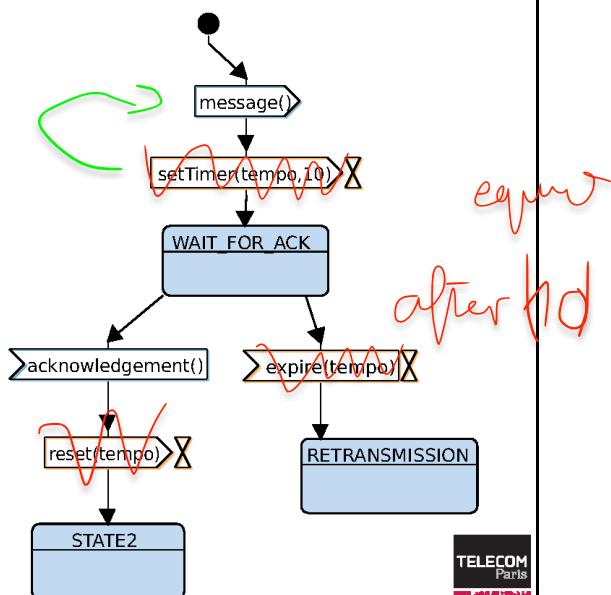
State Machines - Timers (3/3)

"Temporally limited acknowledgement" with timers

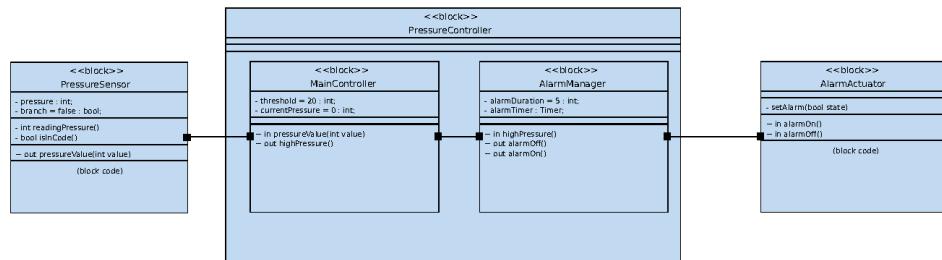
A block instance may take decisions depending on the signal which arrives first: either a "normal" signal or a timer expiration

Question

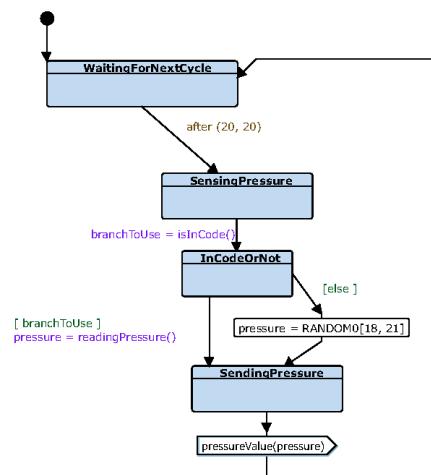
Could we use an *after* clause instead of the *tempo* timer?



Pressure Controller

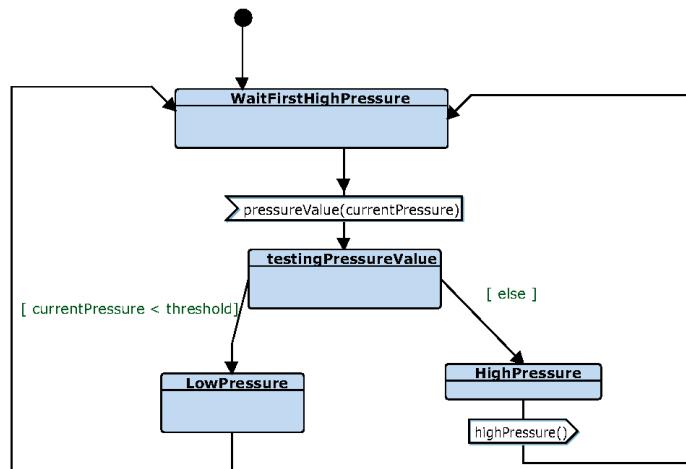


Pressure Controller: States Machines



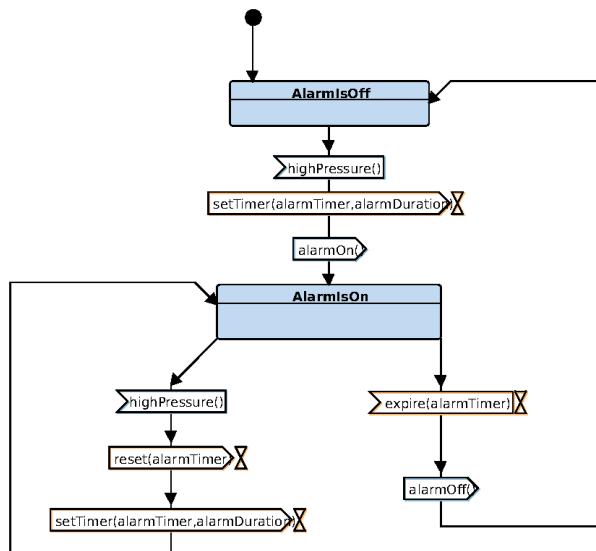
Pressure Sensor

Pressure Controller: States Machines



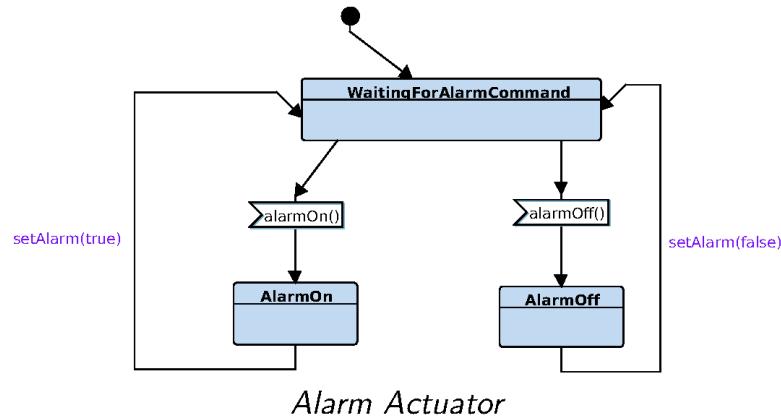
Main Controller

Pressure Controller: States Machines



Alarm Manager

Pressure Controller: States Machines



How to Make "Good" Models?

Practice, Practice and Practice!!!

- Knowledge of various diagrams capabilities
- Accurate understanding of the system to model
- "Reading" your diagrams, reading diagrams of your friends, reading diagrams on Internet
- **Experience is a key factor**

→ Make exercises!