## Outline

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Method</th>
<th>Requirements</th>
<th>(Partitioning)</th>
<th>Analysis</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case Study</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Method</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(Partitioning)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Design</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

1. Requirements
2. Dimensioning Partitioning
   - Simulation Formal verification
3. Analysis
   - Simulation Formal verification
4. Design
   - Simulation Formal verification
5. Deployment
   - Test
Requirement Node

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

- Unique ID. Useful to store requirements in table or to trace them.
- Stereotype (UML extension mechanism).
- Identifier. Should be unique, explicit, yet not too long.
- This text describes the requirement.
- “Kind” classifies the requirement among e.g. functional, non-functional, etc.
**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**

**Case Study**

**Method**

**Requirements**

(Partitioning)

**Analysis**

**Design**

---

**Containment relation**

1. **PressureController**
   - **ID**: 0
   - **Text**: The system shall protect the crew against high pressure
   - **Kind**: Functional

2. **HighPressureDetection**
   - **ID**: 1
   - **Text**: The system shall check the cabin against high pressure
   - **Kind**: Functional

3. **PressureThreshold**
   - **ID**: 2
   - **Text**: The system shall check if the cabin pressure is below a predefined threshold
   - **Kind**: Functional

4. **UseOfPressureSensor**
   - **ID**: 3
   - **Text**: The system shall have a pressure sensor to monitor the pressure of the cabin
   - **Kind**: Functional

5. **CrewInformation**
   - **ID**: 4
   - **Text**: The system shall inform the crew when the cabin has a too high pressure
   - **Kind**: Functional
   - **Risk**: Low

6. **InformWithAlarm**
   - **ID**: 5
   - **Text**: The system shall rely on an alarm to inform the crew about an over pressure in the cabin
   - **Kind**: Functional

7. **AlarmDuration**
   - **ID**: 6
   - **Text**: The over pressure alarm shall last 60 seconds after a high pressure has been detected
   - **Kind**: Functional

8. **OptionalStoringOfPressureValues**
   - **ID**: 7
   - **Text**: All monitored pressure values shall be stored by the software (optional requirement)
   - **Kind**: Functional

9. **RemovableDisk**
   - **ID**: 8
   - **Text**: The system shall store recorded values in a removable disk
   - **Kind**: Functional
**Requirement Diagram - Pressure Controller - Software View**

- **PressureControllerSoftware**
  - **ID=0**
  - Text: "The software shall protect the crew against high pressure"
  - Kind: "Functional"

- **HighPressureDetection**
  - **ID=1**
  - Text: "The software shall check if the cabin pressure is above a predefined threshold"
  - Kind: "Functional"

- **PressureThreshold**
  - **ID=2**
  - Text: "The software shall check if the cabin pressure is below a predefined threshold"
  - Kind: "Functional"

- **UseOfPressureSensor**
  - **ID=3**
  - Text: "The software shall rely on an external pressure sensor to monitor the pressure of the cabin"
  - Kind: "Functional"

- **CrewInformation**
  - **ID=4**
  - Text: "The software is expected to inform the crew when the cabin has a too high pressure"
  - Kind: "Functional"

- **InformWithAlarm**
  - **ID=5**
  - Text: "The software shall trigger an external alarm to inform the crew about an over pressure in the cabin"
  - Kind: "Functional"

- **AlarmDuration**
  - **ID=6**
  - Text: "The over pressure alarm shall last 60 seconds after a high pressure has been detected"
  - Kind: "Functional"

- **OptionalStoringOfPressureValues**
  - **ID=7**
  - Text: "All monitored pressure values shall be stored by the software (optional requirement)"

- **RemovableDisk**
  - **ID=8**
  - Text: "The software shall store recorded values in a removable disk provided with the overall system"

---

**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

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

Important key design parameters
- Speed
- Power Consumption
- Silicon area
- Generation of heat
- Development effort
- ...
Partitioning with the Y-Methodology

- Example: the DIPLODOCUS methodology

Application Modeling

Functions are first modeled independently from the architecture.
Architecture Modeling

Then, architecture is modeled based on generic hardware components: microprocessors, buses, memories, bridges, etc.

Mapping

Functions are then associated to architecture components.
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
- **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

![Use case name](image)

**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
    ![Function inclusion](image)

- **Extension**
  - A function optionally includes another function
    ![Extension](image)

- **Inheritance**
  - A “child” function specializes a “parent” function
    ![Inheritance](image)
Location-Driven Use Case Diagram

Activity Diagram - Syntax

Shows functional flows in the form of succession of actions
Activity Diagram - Pressure Controller

1. readSensorValue(x)
   - If x < threshold, then startAlarm
   - Else, waitFor_60s
     - After 1 second
     - If timerValue == 0, then stopAlarm

2. startAlarm
3. waitFor_60s
4. stopAlarm
5. timerValue = 60
6. timerValue = timerValue - 1
7. After 1 second
8. timerValue = timerValue - 1
9. stopAlarm
Activity Diagram - Pressure Controller

1. `readSensorValue(x)`
2. `writeToDisk_x_currentTime` [ x < threshold ]
3. `startAlarm`
4. `waitFor_60s`
5. `stopAlarm`
6. `writeToDisk_x_currentTime` [ else ]

Diagram:
- Start node
- `readSensorValue(x)`
- Decision node: [ x < threshold ]
- `startAlarm`
- `waitFor_60s`
- `stopAlarm`
- End node
Case Study  Method  Requirements  (Partitioning)  Analysis  Design

## Activity Diagram - Pressure Controller

![Activity Diagram](image)

### Sequence Diagram

- **An actor interacting with a system**

![Sequence Diagram](image)

- **Two interacting "parts" of the system**

![Sequence Diagram](image)
Sequence Diagram - Messages

- **Synchronous communication** (black arrow)

  Sender name  
  \[
  \text{rendez-vous_message}
  \]
  Receiver name

- **Asynchronous communication** (regular arrow)

  Sender name  
  \[
  \text{to_bequeued_message}
  \]
  Receiver name

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**

- Set timer
- Reset timer
- Timer expiration
Sequence Diagram - Pressure Controller

Nominal trace

- PressureSensorDriver
  - pressure(19)
  - pressure19
  - pressure(18)
  - pressure(21)

- PressureMonitoringSoftware
  - pressure(19)
  - pressure19
  - pressure(18)
  - pressure(21)

- AlarmDriver
  - startAlarm
  - (timer=timerAlarm, duration=60)

Advanced trace

- PressureSensorDriver
  - pressure(19)
  - pressure19
  - pressure(18)
  - pressure(21)

- PressureMonitoringSoftware
  - pressure(19)
  - pressure19
  - pressure(18)
  - pressure(21)

- PressureMonitoringAlarmSoftware
  - timerValue = 60
    - (1)
    - timerValue = 59

- AlarmDriver
  - start
  - (timer=timerAlarm)
  - stopAlarm
  - timerValue = 60
Sequence Diagram - Pressure Controller

Nominal trace - version 2 (save before alarm)

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
### Case Study Method Requirements

#### (Partitioning) Analysis

#### Design

**Block Diagram: Syntax of Blocks**

<table>
<thead>
<tr>
<th>&lt;&lt;datatype&gt;&gt;</th>
<th>myType</th>
</tr>
</thead>
<tbody>
<tr>
<td>- flag = true : bool;</td>
<td></td>
</tr>
<tr>
<td>- number : int;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;&lt;block&gt;&gt;</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>- sequenceNumber ...</td>
<td></td>
</tr>
<tr>
<td>- data : myType;</td>
<td></td>
</tr>
<tr>
<td>- method1()</td>
<td></td>
</tr>
<tr>
<td>- method2(int param)</td>
<td></td>
</tr>
<tr>
<td>- out outputSignal(int p...</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

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

**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

- 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 - Guards Diagram]

State Machines - Time Intervals

- **after** clause with a \([T_{\text{min}}, T_{\text{max}}]\) interval

![State Machines - Time Intervals Diagram]

- A transition with no **after** clause has de facto an \(\text{after}(0,0)\) 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 \( \text{size}(\text{FIFO}, \text{inputSignal}) > 0 \)

- **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

The signal declaration contains a formal parameter
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

```
+---------+        +---------+        +---------+
|         |        |         |        |         |
|  WAIT   |        |  STATE1 |        |  STATE2 |        |  STATE3 |
|         |        |         |        |         |        |         |
| signal1(number) | signal2() | signal3(number, flag) |
```

State Machine - Outputs

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

```
<<block>>
Sender2
- flag : bool
- out signal1() 
- out signal2(bool flag)
```

- 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

```
<<block>>
Block0
- x : int;
  - out sig1(int x)
  - out sig2(int x)
<<block>>
Block1
- x : int;
  - in sig1(int x)
  - in sig2(int x)
```

State Machine - Advanced I/O

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

```
<<block>>
Main
- in go_in()
- in done_in(int x)
- out done_out(int x)
<<block>>
T0
- x : int;
<<block>>
T1
- x : int;
<<block>>
T2
- in done_out;
  in go_in
  out done_out;
  out go_out
```

```
T0
  go_out
  WaitForDone
  done_in(x)
T1
  go_in()
  x = 1
  done_out(x)
T2
  go_in()
  x = 2
  done_out(x)
```
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

```
<<block>>
TimedSystem
- tempo : Timer;
~ out message()
~ in acknowledgement()
```

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 is 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

![Diagram of Pressure Controller: States Machines]

Pressure Sensor

Main Controller

![Diagram of Main Controller]
Pressure Controller: States Machines

Alarm Manager

Alarm Actuator
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!