UMLEmb: UML for Embedded Systems
II. Modeling in SysML

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

LabSoC, Sophia-Antipolis, France
Outline

Pressure Controller

Methodology

Requirements

(Partitioning)

Analysis

Design
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

Pressure Controller

Methodology

Requirements

(Partitioning)

Analysis

Design
Overview of the V Cycle

- **Requirements**
- **Dimensioning Partitioning**
- **Analysis**
- **Design**
- **Deployment**
- **Simulation**
  - Formal verification

The V Cycle is a methodology for system development that involves a cycle of analysis and design, followed by implementation and testing.
Outline

Pressure Controller

Methodology

Requirements

(Partitioning)

Analysis

Design
A requirement node identifies a requirement by:

- A unique identifier (so as to achieve tracability)
- A description in plain text
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

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

**HighPressureDetection**
- **ID=1**
- **Text**: The system shall check the cabine against high pressure
- **Kind**: Functional

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

**UseOfPressureSensor**
- **ID=3**
- **Text**: The system shall use a pressure sensor to monitor the pressure of the cabine
- **Kind**: Functional

**CrewInformation**
- **ID=4**
- **Text**: The system shall inform the crew when the cabine has a too high pressure
- **Kind**: Functional

**InformWithAlarm**
- **ID=5**
- **Text**: The system shall rely on an 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 system shall stored recorded values in a removable disk
- **Kind**: Functional
**Requirement Diagram - Pressure Controller - Software View**

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

2. **HighPressureDetection**
   - ID=1
   - Text="The software shall check the cabin against high pressure"
   - Kind="Functional"

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

4. **UseOfPressureSensor**
   - ID=3
   - Text="The software shall rely on a pressure sensor to monitor the pressure of the cabin"
   - Kind="Functional"

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

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

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

8. **RemovableDisk**
   - ID=8
   - Text="The software shall store recorded values in a removable disk"
   - Kind="Functional"

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

Pressure Controller

Methodology

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
- ...
Level of Abstraction

Problematic
- Designers struggle with the complexity of today’s circuits
- 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-Methodology

- Example: the DIPLODOCUS methodology
Application Modeling

Simulation
Static analysis

Application modeling

Architecture modeling

Functions are first modeled independently from the architecture

DSE

mapping

Simulation
Static analysis
Architecture Modeling

Simulation
Static analysis

Application modeling

Mapping

Architecture modeling

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

Simulation
Static analysis

DSE
Mapping

Simulation

Static analysis

Application modeling

Architecture modeling

DSE

Functions are then associated to architecture components

Static analysis

mapping
Outline

Pressure Controller

Methodology

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’s 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

Boundary

Actor

My system

Use case

<<Actor>>

Actor
Use Case Diagram - Pressure Controller - System View

PressureSensor

getPressureInformation

Version2

<<include>>

<<extend>>

PressureMonitoringSystem

monitorAlarm

savePressureValues

Alarm

Storage
Use Case Diagram - Pressure Controller - Software View

PressureSensorDriver

PressureMonitoringSoftware

getPressureInformation

Version2

<<include>>

monitorAlarm

<<extend>>

savePressureValues

AlarmDriver

StorageDriver
**Actors**

- **Syntax 1**: Stickman

  ![Stickman Actor](image)

- **Syntax 2**: `<<Actor>>`

  ```markdown
  `<<Actor>>`
  
  **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]

**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

  ![Diagram of Inclusion](image)

- **Extension**
  - A function optionally includes another function

  ![Diagram of Extension](image)

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

  ![Diagram of Inheritance](image)
Location-Driven Use Case Diagram

- CarAssembly
  - StuttgartFactory
  - MakeFrame
  - SettleWheels
  - BarcelonaFactory
  - Integrate
  - ParisFactory
  - ...
Activity Diagram - Syntax

Shows functional flows in the form of succession of actions

- Activity
- Decision
- Boolean condition
- Fork
- Join
- Connector
Activity Diagram - Pressure Controller

```
readSensorValue(x)
[ x < threshold ]
[ else ]
act
startAlarm
act
waitFor_60s
act
stopAlarm
```
Activity Diagram - Pressure Controller

readSensorValue(x)

[ x < threshold ]

act

WriteToDisk_x_currentTime

[ else ]

act

startAlarm

act

waitFor_60s

act

stopAlarm

[ x < threshold ]
Activity Diagram - Pressure Controller

- readSensorValue(x)
- if \( x < \text{threshold} \):
  - act: WriteToDisk_x_currentTime
  - waitFor_60s
  - act: stopAlarm
  - act: WriteToDisk_x_currentTime
- else:
  - act: startAlarm
  - waitFor_60s
  - act: stopAlarm
Sequence Diagram

- An actor interacting with a system

- Two interacting "parts" of the system
Sequence Diagram - Messages

- **Synchronous communication** (black arrow)

```
Sender_name
rendez_vous_message
Receiver_name
```

- **Asynchronous communication** (regular arrow)

```
Sender_name
to_be_queued_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

Sender

{timer=myTimer, duration=10}

request

ack

{timer=myTimer}

{timer=myTimer, duration=5}

request

{timer=myTimer}

{timer=myTimer}

Receiver
Sequence Diagram - Pressure Controller

Nominal trace

PressureSensorDriver

PressureMonitoringSoftware

AlarmDriver

pressure(19)
president19
pressure(18)
president(21)
startAlarm

{timer=timerAlarm, duration=60}

{timer=timerAlarm}

stopAlarm
Sequence Diagram - Pressure Controller

Advanced trace

PressureSensorDriver - PressureMonitoringSoftware - PressureMonitoringAlarmSoftware - AlarmDriver

PressureSensorDriver

PressureMonitoringSoftware

PressureMonitoringAlarmSoftware

AlarmDriver

timerValue = 60

{1..1}

timerValue = 59

timerValue = 60

Fall 2019

UME - Modeling in SysML
Nominal trace - version 2 (save before alarm)
Sequence Diagram - Pressure Controller

Nominal trace - Version 2 (save after alarm)
Outline

Pressure Controller

Methodology

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 \(\rightarrow\) *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 \(\rightarrow\) *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
**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

- **Initial State (entry point in the state machine)**
- **Initialization action**
- **(Stable) State**
- **The transition has no guard and no action**
- **Choice**
- **Boolean condition**
- **Final state (the block instance ‘dies’)**

- **Boolean condition**
- **[ else]**
- **[ lossy]**

- **STATE1**
- **STATE2**
State Machine - States and Transitions

Note

- No parallelism
- Choices are optional: several guarded - or not guarded - transitions can directly exit a state
State Machines - Guards

- A transition guard contains a *boolean expression* built upon boolean operators and attributes.
# State Machines - Time Intervals

- The *after* clause associates a \([T_{\text{min}}, T_{\text{max}}]\) interval to the transition’s enabling condition.

![State Machine Diagram]

- 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 $\text{size}(\text{FIFO, inputSignal}) > 0$

- Synchronous communication
  - The transition is fired whenever a rendezvous is possible

- Signals can convey parameters
State Machine - Inputs (2/3)

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

```plaintext
<<block>>
System1
- number : int;
~ in signal(int param)
```

The signal reception uses a real parameter, which is an attribute of the block instance that contains the state machine.

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
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

```
x = 5 : int;
~ out sig1(int x)
~ out sig2(int x)
```

```
x = int;
~ in sig1(int x)
~ in sig2(int x)
```

```
x = x + 1
sig1(x)
sig2(x)
sig1(x)
x = x + 1
sig2(x)
Got2
Got1
```
Signals declared by a block may be used by its sub-blocks

- **T0**
  - `~ in go_in()`
  - `~ out go_out()`
  - `~ in done_in(int x)`
  - `~ out done_out(int x)`

- **T1**
  - `T0`
  - `- x : int;`
  - `~ in done_in(int x)`
  - `~ out done_out(int x)`

- **T2**
  - `~ in go_in()`
  - `- x : int;`
  - `~ in done_in(int x)`
  - `~ out done_out(int x)`

**Main**
- `~ in go_in()`
- `~ out go_out()`
- `~ in done_in(int x)`
- `~ out done_out(int 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*?

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

1. **T0**
   - go_out()
   - WaitForDone
   - done_in(x)

2. **T1**
   - go_in()
   - done_out(x)
   - x = 1

3. **T2**
   - go_in()
   - done_out(x)
   - x = 2
State Machine Diagram - Pressure Controller

- Shows the inner functioning of the Controller block instance

Diagram:
- Running
  - highPressure()
  - startAlarm()
  - after (alarmDuration,alarmDuration)
  - stopAlarm()
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
"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

- PressureController
  - MainController:
    - threshold = 20 : int;
    - currentPressure = 0 : int;
  ~ in pressureValue(int value)
  ~ out highPressure()

- AlarmManager:
  - alarmDuration = 5 : int;
  - alarmTimer : Timer;
  ~ in highPressure()
  ~ out alarmOff()
  ~ out alarmOn()

- PressureSensor:
  - pressure : int;
  - branch = false : bool;
  - int readingPressure()
  - bool isInCode()
  ~ out pressureValue(int value)

- AlarmActuator:
  - setAlarm(bool state)
  ~ in alarmOn()
  ~ in alarmOff()
Pressure Controller: States Machines

Pressure Sensor
Pressure Controller: States Machines

Main Controller

- WaitFirstHighPressure
  - pressureValue(currentPressure)
    - [ currentPressure < threshold]
    - [ else ]
      - HighPressure
      - LowPressure
  - [ else ]
    - currentPressure < threshold
    - highPressure()
Pressure Controller: States Machines

Alarm Manager

- **AlarmIsOff**
  - highPressure()
  - setTimer(alarmTimer, alarmDuration)
  - alarmOn()

- **AlarmIsOn**
  - highPressure()
  - expire(alarmTimer)
  - reset(alarmTimer)
  - setTimer(alarmTimer, alarmDuration)
  - alarmOff()
  - alarmOn()
Pressure Controller: States Machines

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!