UMLEmb: UML for Embedded Systems
II. Modeling in SysML

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

LabSoC, Sophia-Antipolis, France
Goals

Learning objective
- Ability to read SysML/AVATAR diagrams
- Knowledge of the SysML/AVATAR syntax to be used during the lab

Content
- Modeling assumptions
- Requirements diagram
- Partitioning
- Use case, sequence and activity diagrams
- Block instance and state machine diagrams
- Educational case study: a pressure controller
Case Study: a Pressure Controlling System

Specification (from the client)

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

Methodology

Requirements

From partitioning to design
Overview of the V Cycle
Outline

Methodology

Requirements

From partitioning to design
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, ...)

**Stereotype**

(UML extension mechanism)

**Requirement Node**

ID=2
Text="The system shall check the cabin against high pressure."
Kind="Functional"

The identifier, which is unique in the requirement diagram, enables traceability

An informal text describes the requirement

Requirement type
- functional
- non functional
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

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

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

- **PDetection**
  - ID=3
  - Text="The system shall compare the pressure with a predefined threshold."
  - Kind="Functional"

- **PressureSensor**
  - ID=4
  - Text="The system shall use a pressure sensor."
  - Kind="Functional"

- **Information**
  - ID=5
  - Text="The system shall make the crew aware of danger."

- **Alarm**
  - ID=6
  - Text="The system shall monitor a sound alarm."
  - Kind="Functional"

- **AlarmDuration**
  - ID=7
  - Text="The system shall monitor an alarm whose duration is equal to 60s."

- **Optional_Storage**
  - ID=1
  - Text="The system shall save the results of measurements."
  - Kind="Functional"

- **RemovableDisk**
  - ID=8
  - Text="The system shall use a removable disk."

**Methodology**: Requirements

**From partitioning to design**

**10/60 Fall 2019**
Outline

Methodology Requirements

From partitioning to design

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
- \( \rightarrow \) 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
Architecture Modeling

Simulation
Static analysis

Application modeling

Architecture modeling

Simulation
Static analysis

DSE

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

Simulation
Static analysis

Application modeling

Architecture modeling

Functions are then associated to architecture components

Static analysis

DSE

mapping
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
Use Case Diagram - Pressure Controller
Actors

- **Syntax 1:** Stickman
  
  ![Stickman Actor]

- **Syntax 2:** «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

  ![Inclusion Diagram]

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

  ![Extension Diagram]

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

  ![Inheritance Diagram]
Location-Driven Use Case Diagram

CarAssembly

StuttgartFactory

MakeFrame

BarcelonaFactory

SettleWheels

ParisFactory

Integrate

...
Activity Diagram - Syntax

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

**Act** Initialize

**Act** analyzePressure

- pressure > threshold
- pressure <= threshold

**Act** informSupervisor
Sequence Diagram

- An actor interacting with a system

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

- **Synchronous communication** (black arrow)

  ![Synchronous Communication Diagram](image)

- **Asynchronous communication** (regular arrow)

  ![Asynchronous Communication Diagram](image)
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

![Relative dates diagram](image)

### Absolute date

![Absolute date diagram](image)
Sequence Diagram - Time (2/2)

- **Timers**

  - **Set timer**
  - **Reset timer**
  - **Timer expiration**

Diagram:

```
Sender
\[\{timer=myTimer\}\]
\{timer=myTimer, duration=5\}
\{timer=myTimer, duration=10\}
request
ack
Receiver
\{timer=myTimer\}
\{timer=myTimer\}
```

- **From partitioning to design**
Sequence Diagram - Pressure Controller

- Shows how the system and the actors communicate over time
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
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’)**

**Diagram:**
- Initial state labeled as `STATE1` with a transition to `STATE2`.
- Transition from `STATE2` to `STATE1` with a boolean condition `[else]`.
- Transition from `STATE1` to itself with a boolean condition `[lossy]`.

**Annotations:**
- `lossy = false` on the transition from `STATE1` to `STATE2`.

**Legend:**
- Red dashed line indicates transitions.
- Green boxes provide descriptions of states and actions.
State Machine - States and Transitions

**Note**
- No parallelism
- Choices are optional: several guarded - or not guarded - transitions can directly exit a state

![Diagram showing state transitions]

- lossy = false
- [else]
- [lossy]
State Machines - Guards

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

![Diagram of state machine with guards and actions](image-url)
State Machines - Time Intervals

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

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

- **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.

```
<<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.
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

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

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.
Synchronous Communications

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

```
<<block>>
Block0
- x = 5 : int;
~ out sig1(int x)
~ out sig2(int x)
<<block>>
Block1
- x : int;
~ in sig1(int x)
~ in sig2(int x)
```
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

```
<<block>>
Block0
- x = 5 : int;
~ out sig1(int x)
~ out sig2(int x)
<<block>>
Block1
- x : int;
~ in sig1(int x)
~ in sig2(int x)
```

```
sig1(x)  
\rightarrow  x = x + 1  
sig1(x)
\rightarrow  x = x + 1  
sig2(x)
```

```
Got2
\rightarrow  sig2(x)
\rightarrow  Got1
\rightarrow  sig1(x)
```
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

```
<<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
```

```
go_out()
WaitForDone
```

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

```
T0
done_in(x)
```

```
T1
x = 1
done_out(x)
```

```
T2
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

```
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()
```

```
Timer declaration
```
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
"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?
Another Model for 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)

(block code)

AlarmActuator
- setAlarm(bool state)
~ in alarmOn()
~ in alarmOff()

(block code)
Another Model for Pressure Controller

```
waitingForNextCycle

after (1,1)

sensingPressure

branch = isInCode()

[ branch ]

pressure = readingPressure()

[ else ]

sendingPressure

pressure = RANDOM0[19, 21]

pressureValue(pressure)
```

Pressure Sensor
Another Model for Pressure Controller

Main Controller
Another Model for Pressure Controller

Alarm Manager

Diagram: State transitions and actions for the alarm state machine.
Another Model for Pressure Controller

Alarm Actuator

WaitingForAlarmCommand

setAlarm(true)

alarmOn()

setAlarm(false)

alarmOff()
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!