Modèle et génération automatique de code

Alexandre Chapoutot

ENSTA Paris

2023-2024

Part I

Lecture 1

Introduction

Anatomy of an embedded system

Embedded systems are made of different components which highly interact together.

The jobs of the designer and the programmer are:

- Design an algorithm to control a physical process
- Implement this algorithm on a given hardware

Remark: we deal with two different worlds

- continuous-time evolution for the plant.
- discrete-time (periodic sampling) evolution for the software.

Ingredients for the design of embedded systems

The challenges

1. Safety critical systems (e.g. x-by-wire systems)
2. Complex functions to implement
3. Reduction of the time-to-market design

Ingredients for the design of embedded systems

The challenges

1. Safety critical systems (e.g. x-by-wire systems)
2. Complex functions to implement
3. Reduction of the time-to-market design

Solution for problem 1

For the past decades some **safety standards** have been written such as

- avionic: DO178B/C
- automotive: ISO 26262

Consequence:

- the life cycle of development is codified;
- definition of system level of integrity (SIL or ASIL);
- a set of rules have to be followed to certify the SIL, *e.g.* mc/dc test coverage.

This is **a very long and costly process**.

Ingredients for the design of embedded systems

The challenges

1. Safety critical systems (e.g. x-by-wire systems)
2. Complex functions to implement
3. Reduction of the time-to-market design

Solution for problem 2

Increase the level of abstraction to describe the systems.

Initially software was written in assembly language

Now, more and more the development uses the **model-based design** paradigm.

Stay independent of a particular architecture as long as possible to simplify the reasoning and to increase reusability.

Ingredients for the design of embedded systems

The challenges

1. Safety critical systems (e.g. x-by-wire systems)
2. Complex functions to implement
3. Reduction of the time-to-market design

Solution for problem 3

Increase the use of **automatizing** in the process of development.

It is possible with the model-based design because we can make early simulation at the beginning of the cycle of development.

Moreover, the increase of productivity is possible if the **process of development is well defined**.

Challenge for the industry

To emphasize the challenges for the industry, we recall:

- In 2007: Daimler Chrysler recalls 62369 vehicles to reprogram brake systems.
- Between 2009 and 2010: Toyota recalls around $2M$ vehicles to fix a acceleration pedal problem.

And last but not least, Volvo and

- its crash avoidance system in the S60 vehicle.
- its pedestrian crash avoidance system in the V60 vehicle.

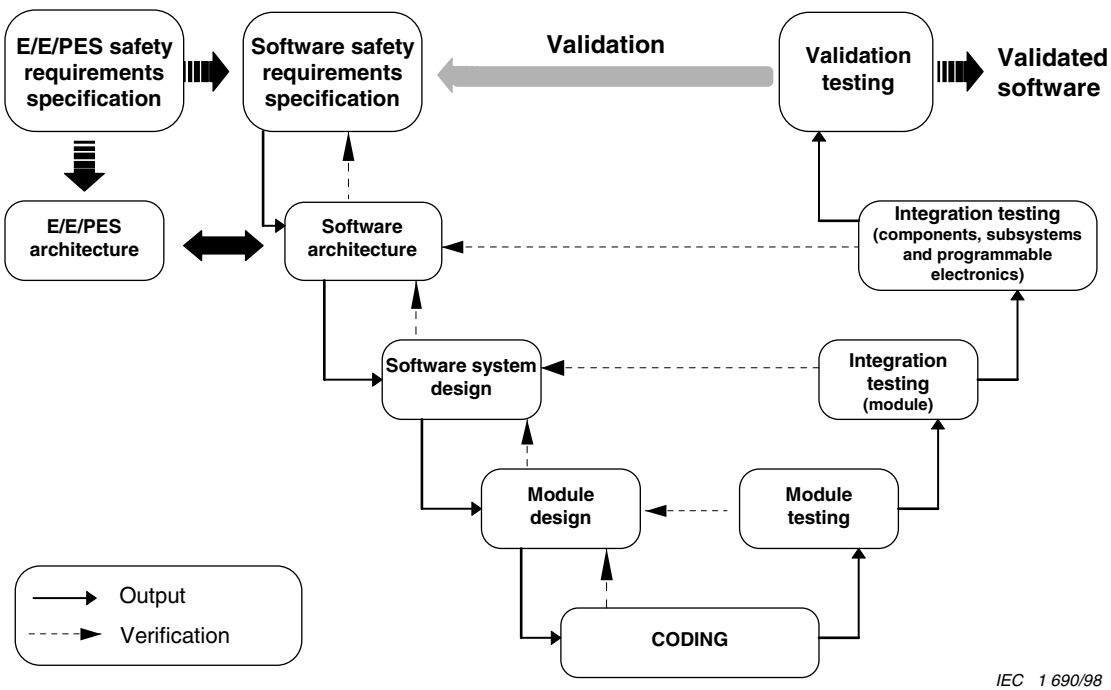
Conclusion

The design of embedded systems is highly difficult and critic for industry.

Cycle of development

Goals

- Split the system into sub-systems.
- Specification and conception of the sub-systems.
- Integration of all the components: gather sub-systems and validate the behaviors.



Design of an embedded system

The classical flow of conception is based on the V cycle which is paper intensive.
Model-based design uses "computers" to help designing the embedded systems.

Difficulties of the model-based design

- **Modelling:** Specification and simulation
- **Validation:** Proof of properties

Main technical issues

- Integration of models mixing discrete/continuous behaviors
- Numerical approximations

Remark: two kinds of model-based design

- System level with for example SysML or AADL
- **Component level** with Simulink/Stateflow or Lustre/SCADE

Safety Standard

Definition of system

A system is a **combination** of components which **act together** to produce a **result**.

Service delivered by the system: its behavior as it is perceived by its user(s) or an other system.

- A **failure** is a **temporary or permanent** deviation of the delivered service.
- A failure is due to one or more **errors**.
- An error comes from one or more **faults**.

The safety standards organized the failures such that:

- the **severity** of consequences.
- the **frequency** of occurrences.

A system is critic if consequences of a failure can be catastrophic for human or for the mission.

Dependability

Ability of a system to deliver service that can be justifiably be trusted. (definition due to J.C. Laprie)

Features:

- **Availability:** readiness for correct usage.
- **Reliability:** continuity of correct service.
- **Safety:** absence of catastrophic consequences on the user(s) and the environment.
- **Confidentiality:** Absence of unauthorized disclosure of information.
- **Integrity:** absence of improper system alterations.
- **Maintainability:** ability for a process to undergo modifications and repairs.

We speak about RAMS which stand for Reliability, Availability, Maintainability, Safety. (FMDS en francais)
Security only deals with: Availability, Confidentiality and Integrity aspects.

IEC61508 standard

This standard integrates the reliability into the cycle of development:

- International standard: CEI (Commission Electrotechnique Internationale)
- First edition in 1998
- Related to Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE system).

It is made of 7 parts

- Part 1 : global prescriptions.
- Part 2 : prescriptions related to the hardware of E/E/PE systems.
- **Part 3 : prescriptions related to software.**
- Part 4 : definitions and abbreviations.
- Part 5 : examples of the methods used to determine the safety integrity level.
- Part 6 : guidelines for the application of Part 2 and Part 3.
- Part 7 : overview of the methods that may be used.

Goals of the IEC61508 standard

IEC61508 standard is a **general approach** of all the steps related to the lifecycle of the safety of E/E/PE systems.

Idea : the safety is obtained by combining safety systems:
global approach of the safety and
independence of the domain of activity.

In order to facilitate the definition of new standards for particular areas.
For example,

- Railway: IEC 50129
- Nuclear: IEC 61513/60880
- Automotive: ISO 26262
- etc.

Safety integrity level

The standard uses the notion of **Safety Integrity Level** (SIL) in order to define the targeted level of safety of a system.

It adopts **risk-based** approach to determine the safety integrity levels.

It sets **quantitative objectives** (a lower limit) for the measures of the failures of systems which are **related to** the SIL.

For example, the number of accepted failures in function of the SIL.

SIL	Continuous mode	On demand mode
1	$10^{-5} > \lambda \geq 10^{-6}$	$10^{-1} > \text{PFD} \geq 10^{-2}$
2	$10^{-6} > \lambda \geq 10^{-7}$	$10^{-2} > \text{PFD} \geq 10^{-3}$
3	$10^{-7} > \lambda \geq 10^{-8}$	$10^{-3} > \text{PFD} \geq 10^{-4}$
4	$10^{-8} > \lambda \geq 10^{-9}$	$10^{-4} > \text{PFD} \geq 10^{-5}$

- λ is the probability of failure per hour, e.g. $\lambda = 10^{-9}$ is equivalent to one failure every 1 billion of hours.

Evaluation of SIL: risk graph

Some evaluation methods are given in IEC61508 standard Part 5.
In particular, the method based on **risk graphs** (Appendix D).

In this method, the **risk** R is defined by:

$$R = C \times F \times P \times W$$

Where the 4 parameters are:

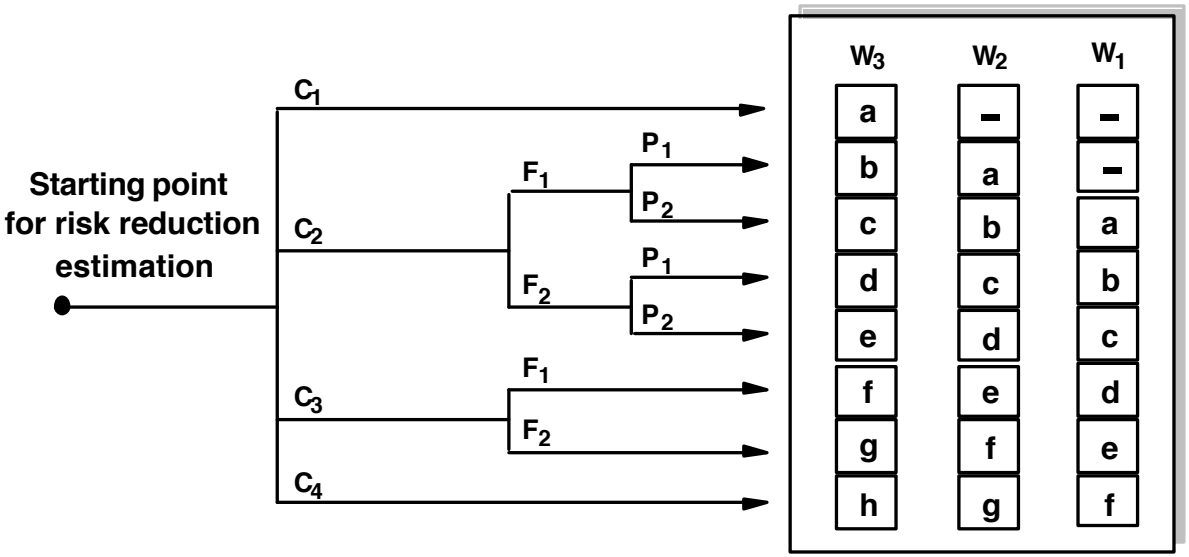
- C : Consequence risk parameter
- F : Frequency and exposure time risk parameter
- P : Possibility of avoiding hazard risk parameter.
- W : Probability of the unwanted occurrence.

Evaluation of SIL: risk graph

Classification of parameters

Consequences	
C1	Minor injury
C2	Serious permanent injury to one or more persons; death to one person
C3	Death to several people
C4	Very many people killed
Frequency of, and exposure time in, the hazardous zone	
F1	Rare to more often exposure in the hazardous zone
F2	Frequent to permanent exposure in the hazardous zone
Possibility of avoiding the event	
P1	Possible under certain conditions
P2	Almost impossible
Probability of the unwanted occurrence	
W1	A very slight probability
W1	A slight probability
W3	A relatively high probability

Evaluation of SIL: risk graph



a, b, c, d, e, f, g, h represent the necessary minimum risk reduction. The link between the necessary minimum risk reduction and the safety integrity level is shown in the table.

C = Consequence risk parameter

F = Frequency and exposure time risk parameter

P = Possibility of avoiding hazard risk parameter

W = Probability of the unwanted occurrence

a, **b**, **c** ... **h** = Estimates of the required risk reduction for the SRSs

Necessary minimum risk reduction	Safety integrity level
-	No safety requirements
a	No special safety requirements
b, c	1
d	2
e, f	3
g	4
h	An E/E/PE SRS is not sufficient

IEC 61508 Standard Part-3

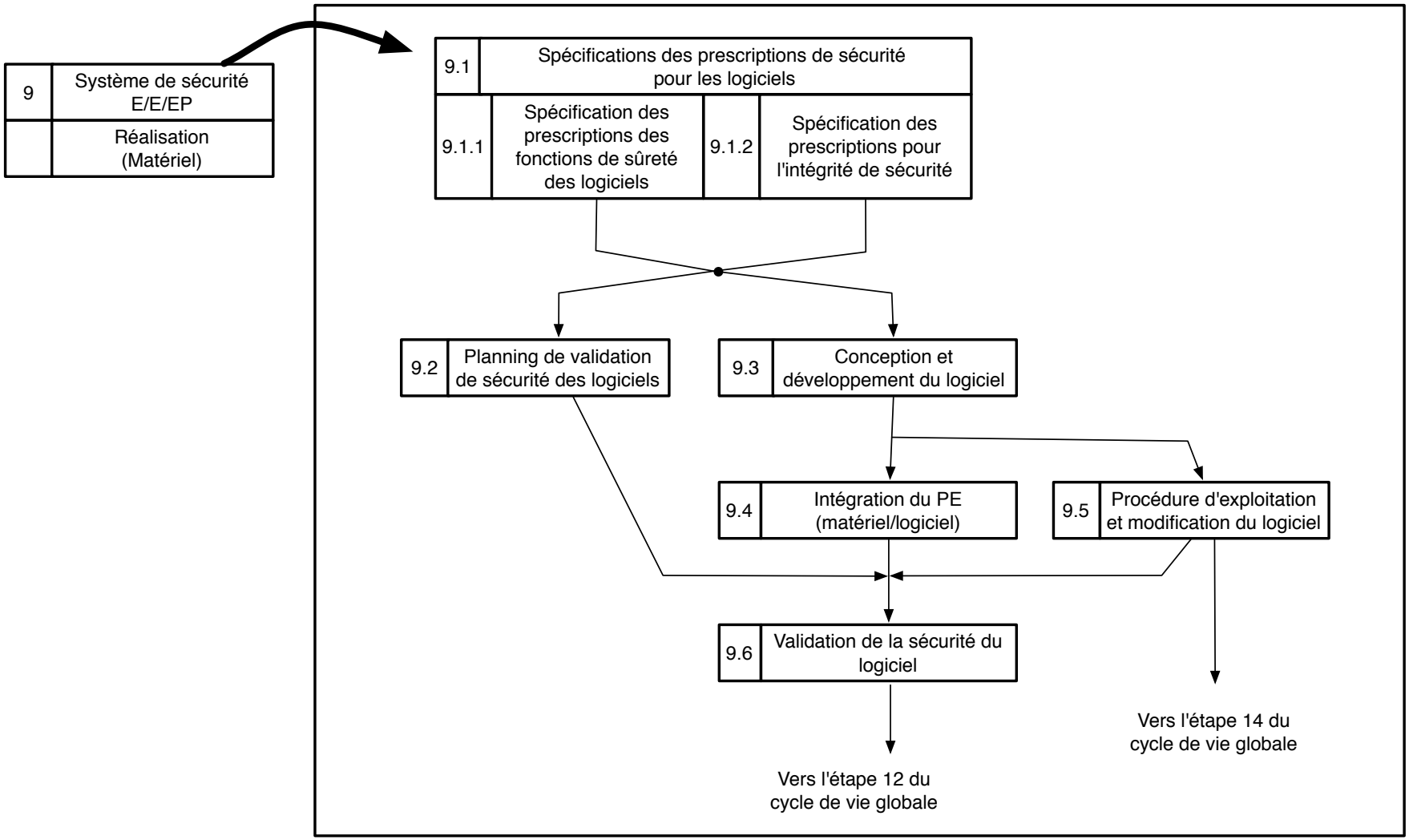
For all software
forming part of a safety related system
or **used** to develop a safety related system.

This part establishes requirements for safety lifecycle phases and activities which shall be applied during the **design** and **development** of the safety-related software.

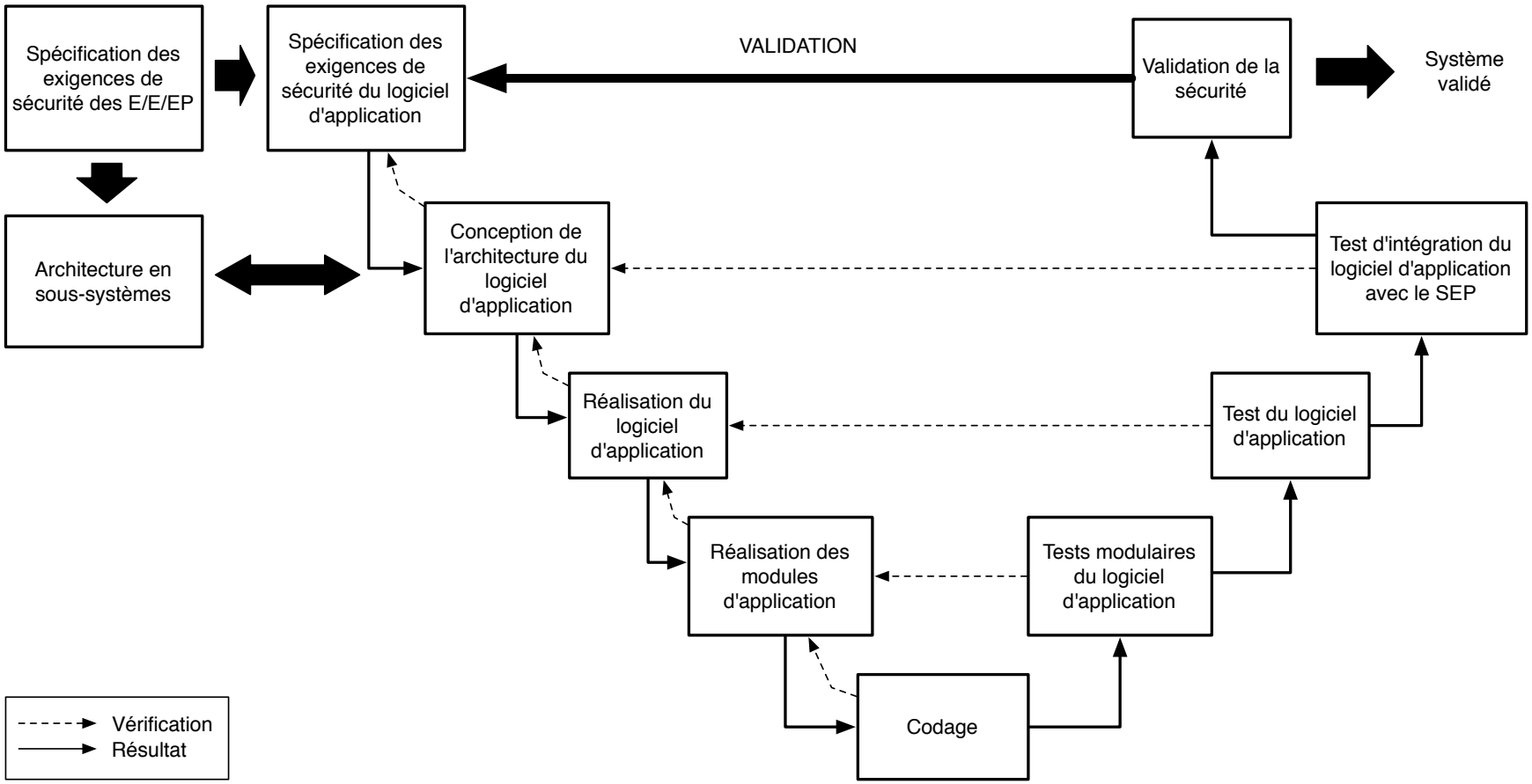
Important features of the lifecycle (design):

- **Step** 9.2 : definition of software specifications from hardware specifications.
- **Step** 9.2 : validation plan (in parallel of the development).
- **Step** 9.3 :
  - Definition of the software architecture and the module specifications.
  - Definition of integration tests (software/hardware, software/software).
  - List of development tools.
- **Step** 9.4 : Results of integration tests.
- **Step** 9.6 : Results of the validation.

Lifecyle of the software (design)



Cycle of development software



- Each step must be validated!

  a set of documents must be given to certification authority.

Recommendations for Step 9.3 (development)

Design tools

Techniques	SIL1	SIL2	SIL3	SIL4
Right programming language	HR	HR	HR	HR
Strong typed programming language	HR	HR	HR	HR
Subset of programming language	–	–	HR	HR
Certified tools (experienced)	R (HR)	HR	HR	HR
Certified compilers (experienced)	R (HR)	HR	HR	HR
Certified/experienced software libraries	R	HR	HR	HR

Recommendations for Step 9.3 (development)

Programming aspects

Techniques	SIL1	SIL2	SIL3	SIL4
Defensive programming	–	R	HR	HR
Modular approach	HR	HR	HR	HR
Coding rules	R	HR	HR	HR
Structured programming	HR	HR	HR	HR
Verified/experienced software libraries	R	HR	HR	HR

And also:

- no dynamical memory;
- limited use of pointers, recursive functions and/or interruptions;
- no goto, ...

Summary

Design of critical embedded systems is **strongly codified** by safety standards.

Always remember that the **cost of the certification** is around 30% to 50% of the global budget of a project.

Tools and design methods which reduce the time-to-market and/or reduce the cost of certification is seek by the industry.

Current strategy

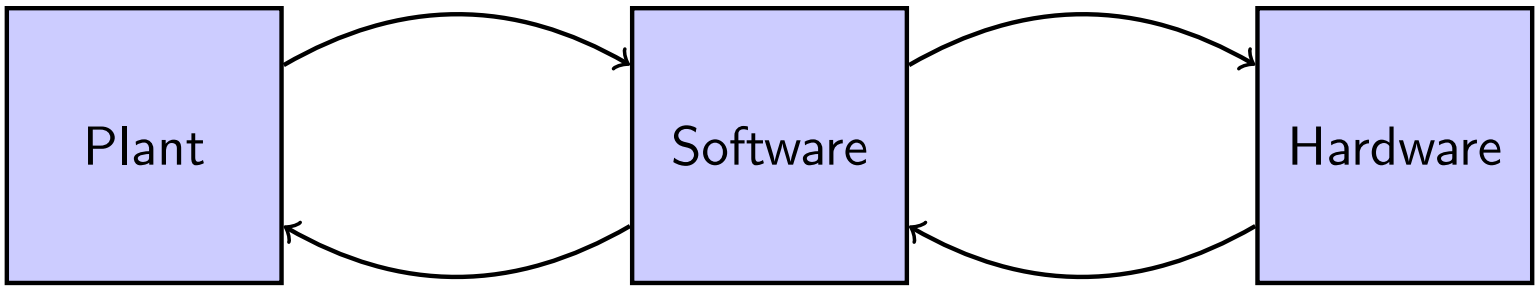
Model-based design: Lustre/SCADE and Matlab/Simulink/Stateflow.

- Important reuse of software;
- Automatic code generation;
- Early error detection: simulation and/or formal methods.

Model-Based Cycle of Development

Anatomy of an embedded system

Embedded systems are made of different components which highly interact together.



The jobs of the designer and the programmer are:

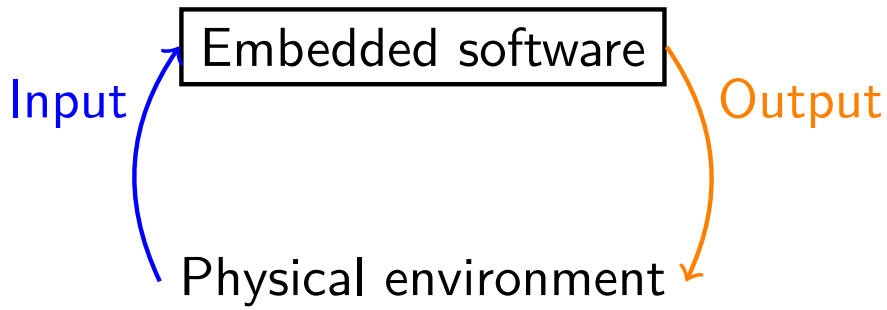
- Design an algorithm to control a physical process
- Implement this algorithm on a given hardware

Remark: we deal with two different worlds

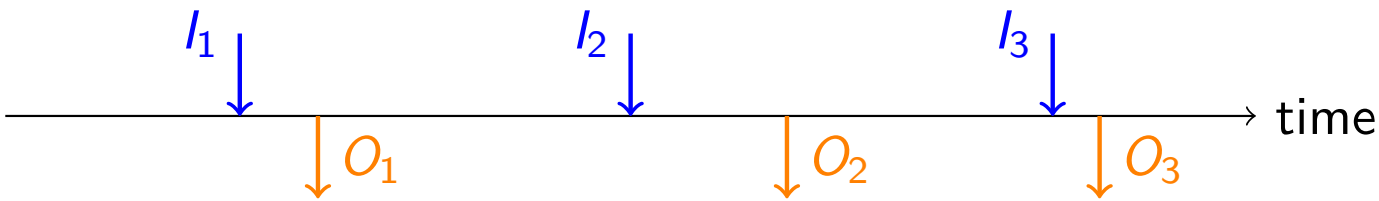
- continuous-time evolution for the plant.
- discrete-time (periodic sampling) evolution for the software.

Reactive software

- **Embedded software** are also known as **reactive programs**: they continuously produce outputs in response to inputs coming from the physical environment.



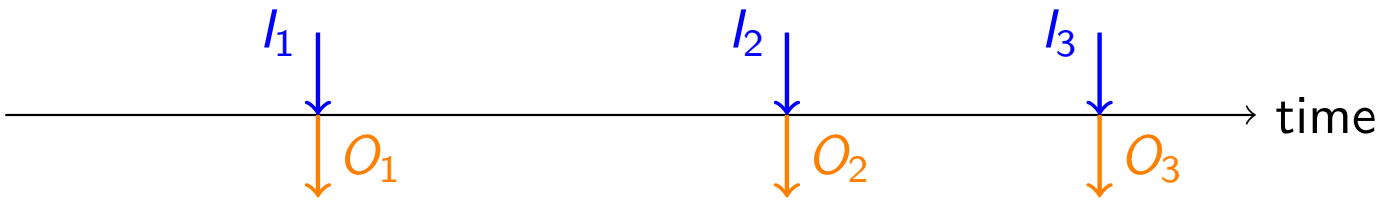
- The execution of embedded software is described by **discrete-time dynamics** *i.e.* it is a sequence of reactions.



- Ideally we should have that:
  - Output O_i should be emitted before input I_{i+1} and no important input I_i is missed (**real-time constraints**)
  - The software is **deterministic**: same input produces same output.
  - **A finite amount of memory** is used.

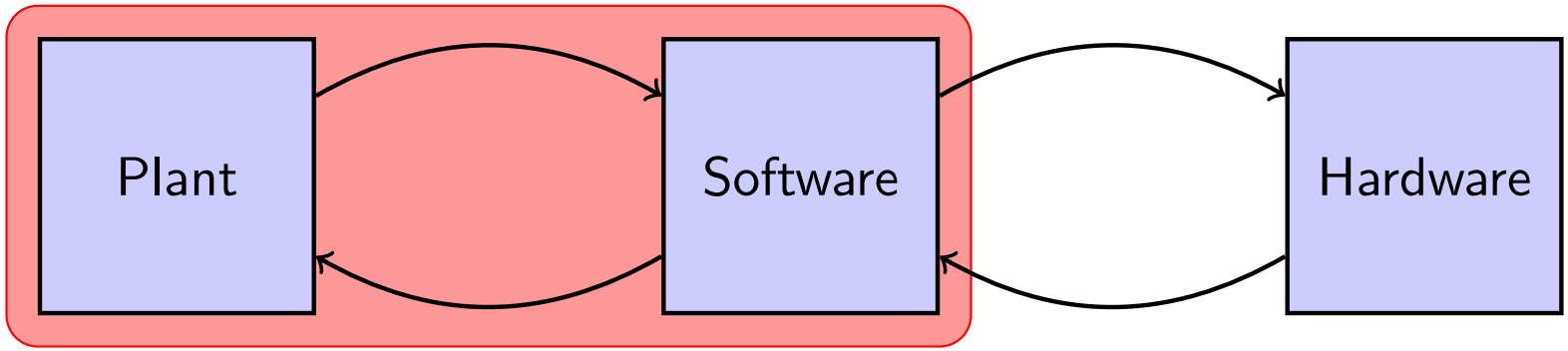
An ideal abstraction: synchronicity

- The execution of embedded software is described by **discrete-time dynamics** *i.e.* it is a sequence of reactions.
  We assume that the computation time is zero



- **Conceptually**
  - Output are produced infinitly quickly
  - All the computation are done in parallel
- **Verification of the hypothesis**
  - Compute WCET and check that input are not faster than WCET

Integration of the software with the environment



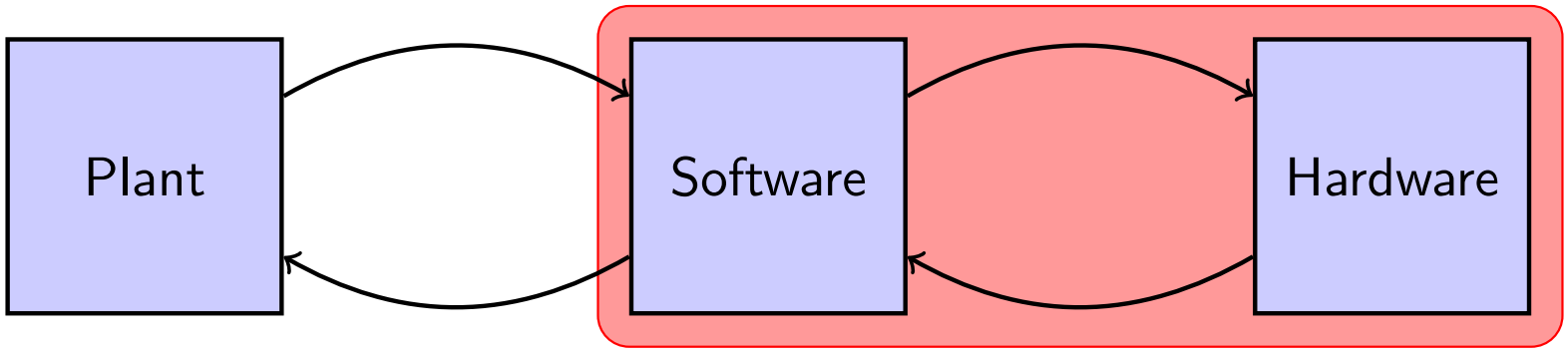
The problem is to define a control/command software with respect to a particular physical environment.

Problematic: How to focus on the function without taking into account the implementation details.

Idea:

- Working with both a model of software and the physical environment.
- Furthermore, with well-defined models we can reason on them.

Integration of the software with the hardware

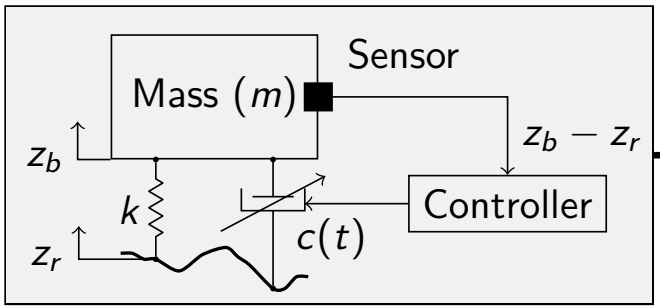


Problematic: to make the software and the hardware working together.

Idea:

- Automatic code generation, parametrized with hardware taget features.
- Furthermore, well-defined language and compilation processes allow to guarantee some good properties as determinism, boundness of memory consumption.

Classical design and validation methodology: automotive

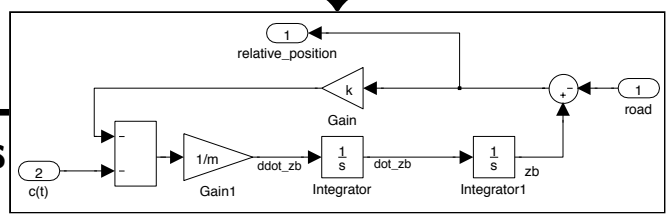


Identification

Mathematical model

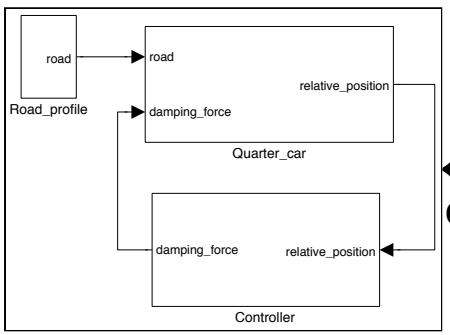
$$m\ddot{x} + c\dot{x} + kx = u$$

Computer description



Theoretical and experimental studies

Controller definition



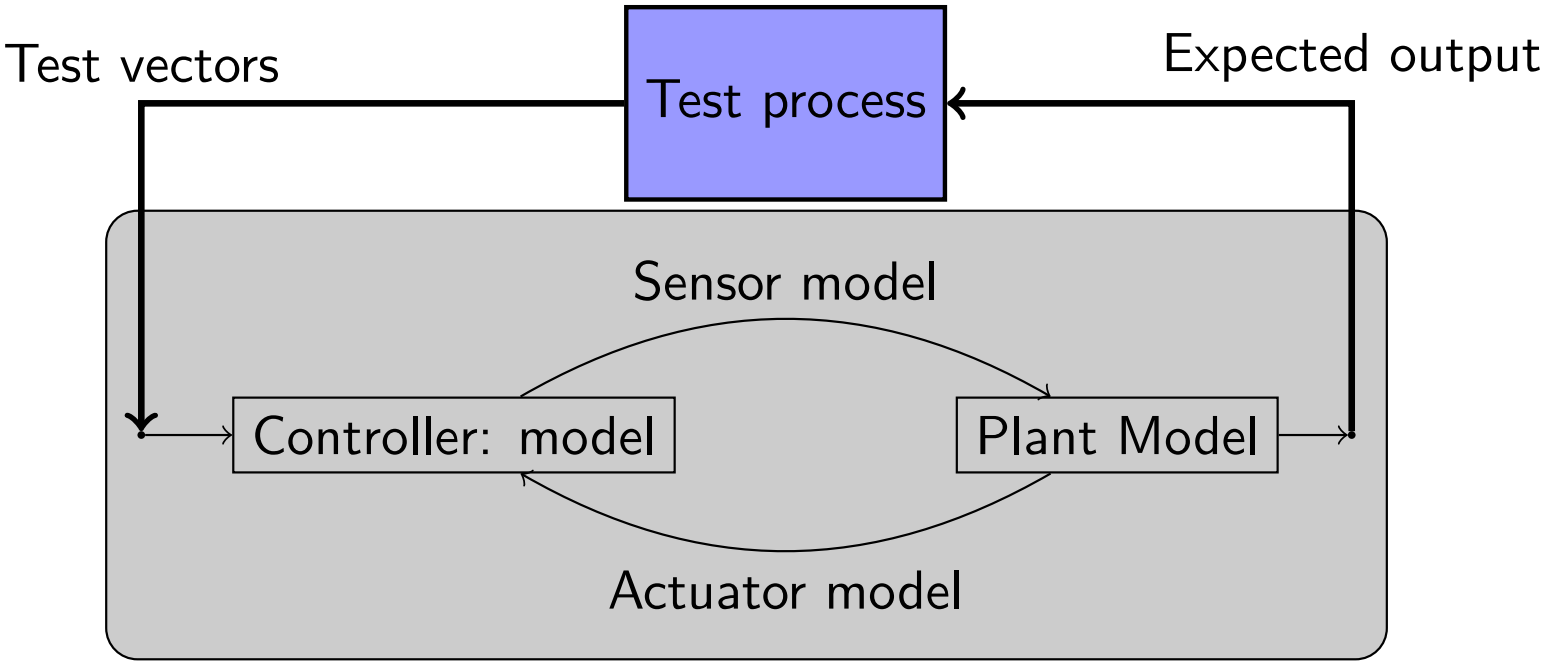
Remarks

- **Mathematical models** are an approximated descriptions of **physical systems**.
- The **computer descriptions** are studied with numerical tools.

Classical design and validation methodology: automotive

MIL: Model in the loop

Definition of a mathematical model of the `plant` and the `controller`.



Simulation platform: Matlab/Simulink/Stateflow

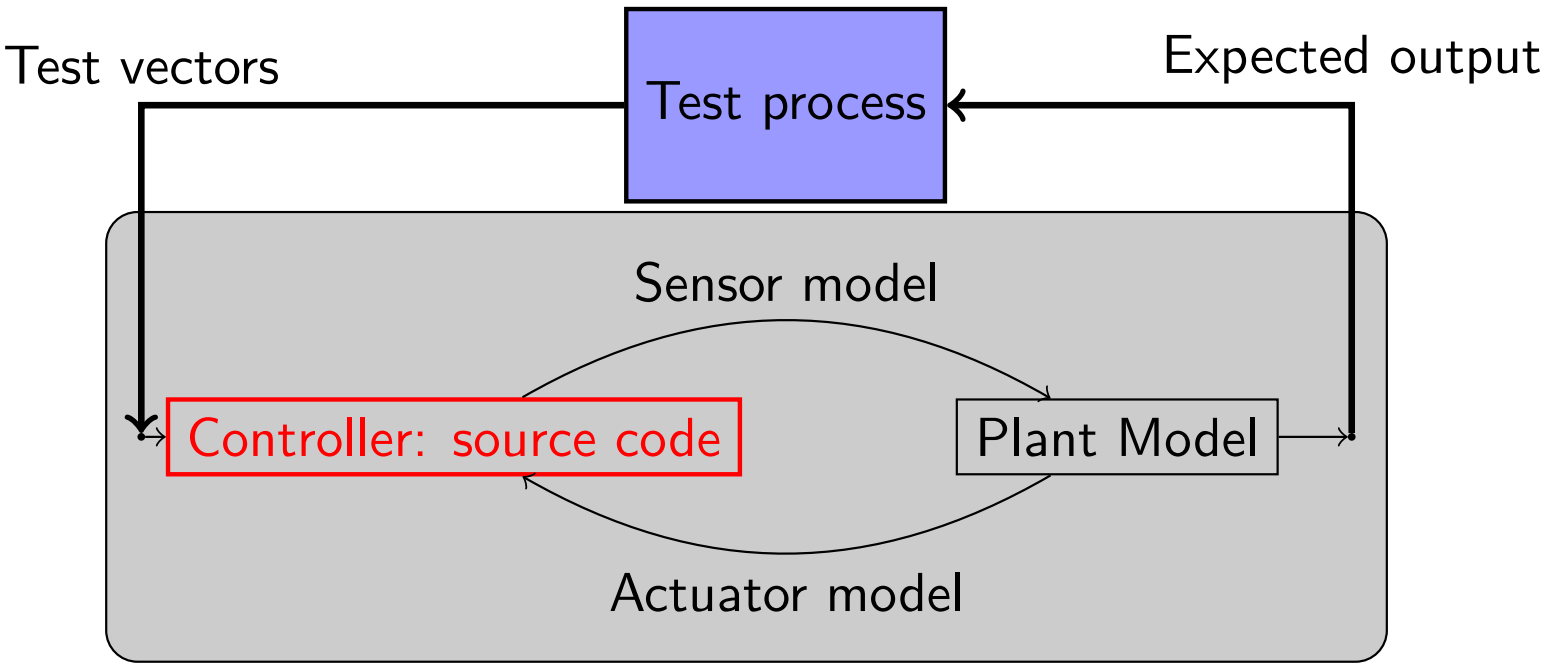
Aim of the testing activity

- Does the controller fulfil the specification?
- The set of tests will be used as "an oracle" in the next steps.

Classical design and validation methodology: automotive

SIL: Software in the loop

Implementation of the `controller` in a target language.



Simulation platform: M/S/S

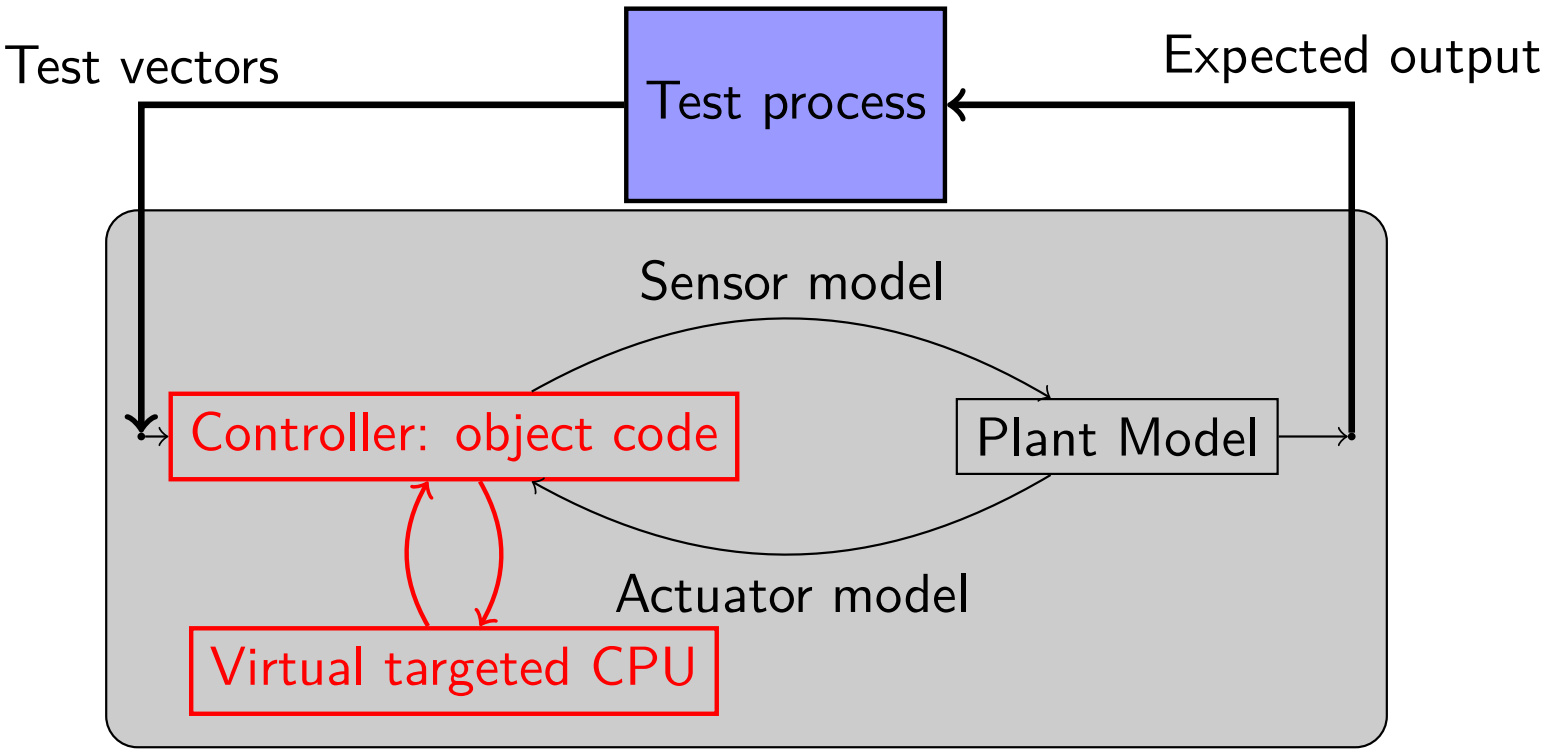
Aim of the testing activity

- Do the hand-written or generated code still fulfil the specification?

Classical design and validation methodology: automotive

PIL: Processor in the loop

Compilation of the controller and execute it on a virtual processor.



Simulation platform: M/S/S

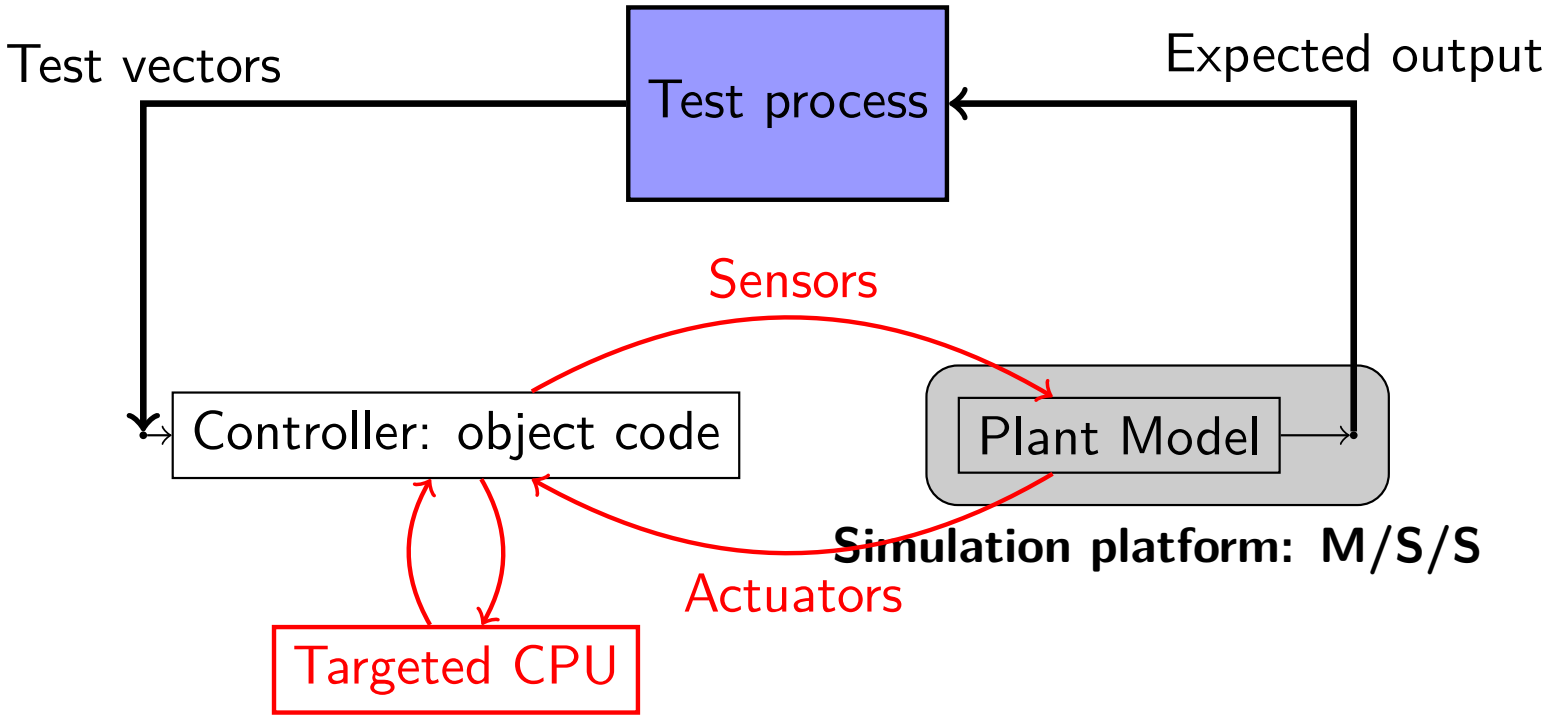
Aim of the testing activity

- Does the low-level implementation still fulfil the specification?

Classical design and validation methodology: automotive

HIL: Hardware in the loop

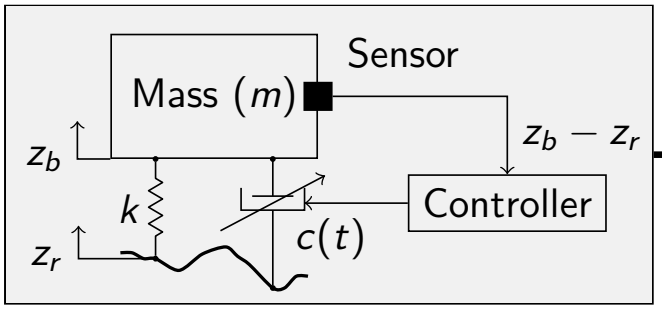
Put the software on the real hardware target with sensors and actuators.



Aim of the testing activity

- Does the implemented solution still fulfil the specification?

Classical design and validation methodology: airplane

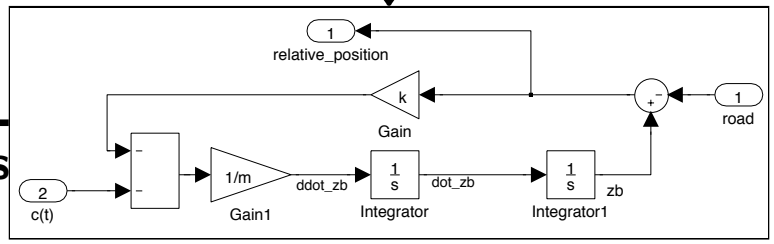


Identification

Mathematical model

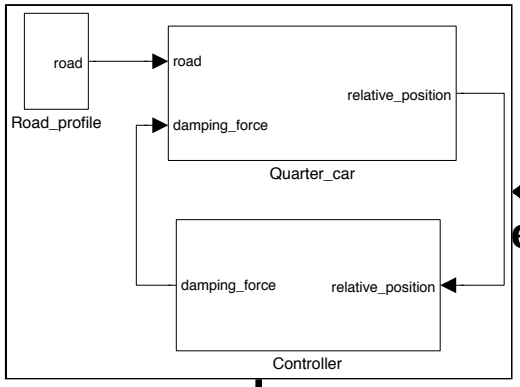
$$m\ddot{x} + c\dot{x} + kx = u$$

Computer description



Theoretical and experimental studies

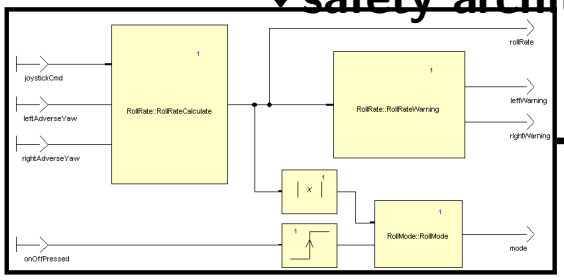
Controller+Plant



safety architecture

SCADE translation

KCG compiler



Controller discrete model

```
/* $*********** KCG Version 6.0.0 (build i10) **********
   Command line: -node RollControl::RollControl -target_dir . kcg_xml_filter_out
   Generation date: 2021T12:08:08
*************************************************$ */
#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "RollControl_RollControl.h"
void RollControl_reset_RollControl(outC_RollControl_RollControl *outC)
{
  /* 1 */
  RollMode_reset_RollMode(&outC->Context_RollMode);
  /* 1 */
  RisingEdge_reset_digital(&outC->Context_RisingEdge);
}
void RollControl_RollControl(
  inC_RollControl_RollControl *inC,
  outC_RollControl_RollControl *outC)
{
  /* RollControl::RollControl::_L16 */
  kcg_bool _L16;

  /* 1 */
  RisingEdge_digital(inC->onOffPressed, &outC->Context_RisingEdge);
  _L16 = outC->Context_RisingEdge.RE_Output;
  outC->rollRate = /* 1 */
    RollRateCalculate_RollRate(
      inC->joystickCmd,
      inC->leftAdverseYaw,
      inC->rightAdverseYaw);
```

Controller implementation

Model-based design and cycle of development

Process that generates the life-cycle data	Type #1	Type #2a	Type #2b	Type #3a	Type #3b
System Requirement and System Design Processes	System requirements allocated to software	Higher-level Requirements	Higher-level Requirements	Higher-level Requirements	Higher-level Requirements
			Design Model		
Software Requirement and Software Design Processes	Higher-level Requirements (= Software high-level requirements)	Design Model		Specification Model (= Higher-Level Requirements for Design Model.)	Specification Model
	Design Model			Design Model	ED-12B / DO-178B Design Description
Software Coding Process	Source Code	Source Code	Source Code	Source Code	Source Code

Problem: add confidence in the built software w.r.t. requirements.
Written software must be safe and so we require it is:

- readable
- deterministic
- without ambiguity

Model-based design approach

Pros

- Ability to **focus on functionality** without taking into account low-level implementation details.
- Ability to **early simulate the system** to found errors or to complete the specifications.
- Ability to **automatically generate code** reducing the introduction of translation errors.
- Ability to **continuously validate** each step of the cycle of development.

Challenges

- **Modelling the physical environment** with enough details.
- **Modelling the software and its interactions** with the `physical environment` and the `hardware`.