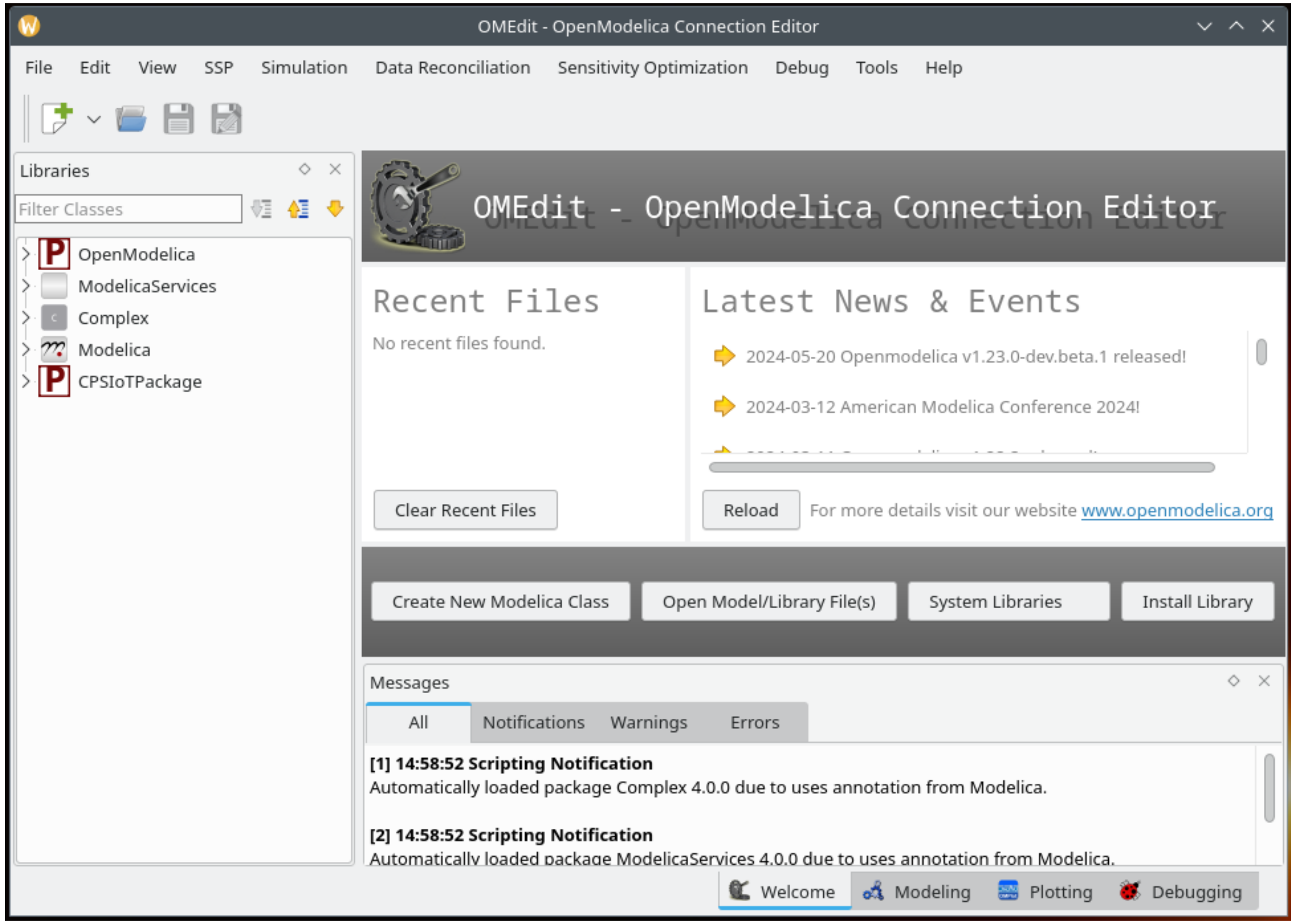
Installing the tool:
OpenModelica Connection Editor (OMEdit)

Download from:
https://openmodelica.org/#



No recent files found.

2024-05-20 Openmodelica v1.23.0-dev.beta.1 released!

2024-03-12 American Modelica Conference 2024!

For more details visit our website www.openmodelica.org

[1] 14:58:52 Scripting Notification
Automatically loaded package Complex 4.0.0 due to uses annotation from Modelica.

[2] 14:58:52 Scripting Notification
Automatically loaded package ModelicaServices 4.0.0 due to uses annotation from Modelica.

Download the tutorial package from the handout website!!!!

mem4csd.telecom-paristech.fr

go to Training Schools > Summer School 2024 > OpenModelica

Modeling a Cruise Control System using OpenModelica and Verifying Safety Requirements using UPPAAL

Rakshit Mittal[1], Hans Vangheluwe[1], Rizwan Parveen[2]

[1]University of Antwerp – Flanders Make, Belgium

[2]Telecom Paris, France

2 hands-on tutorials with foundations in Multi-Paradigm Modeling

Case Study: Adaptive Cruise Control System (ACCS)

1a: Modeling the ACCS using OpenModelica

Rakshit Mittal[1], Hans Vangheluwe[1]

1b: Verifying ACCS Safety Requirements using UPPAAL

Rizwan Parveen[2]

2a: Modeling and Analyzing the Architecture of the ACCS controller using AADL

Dominique Blouin[2], Anish Bhobe[3]

2b: Synthesizing Code for the ACCS controller using RAMSES

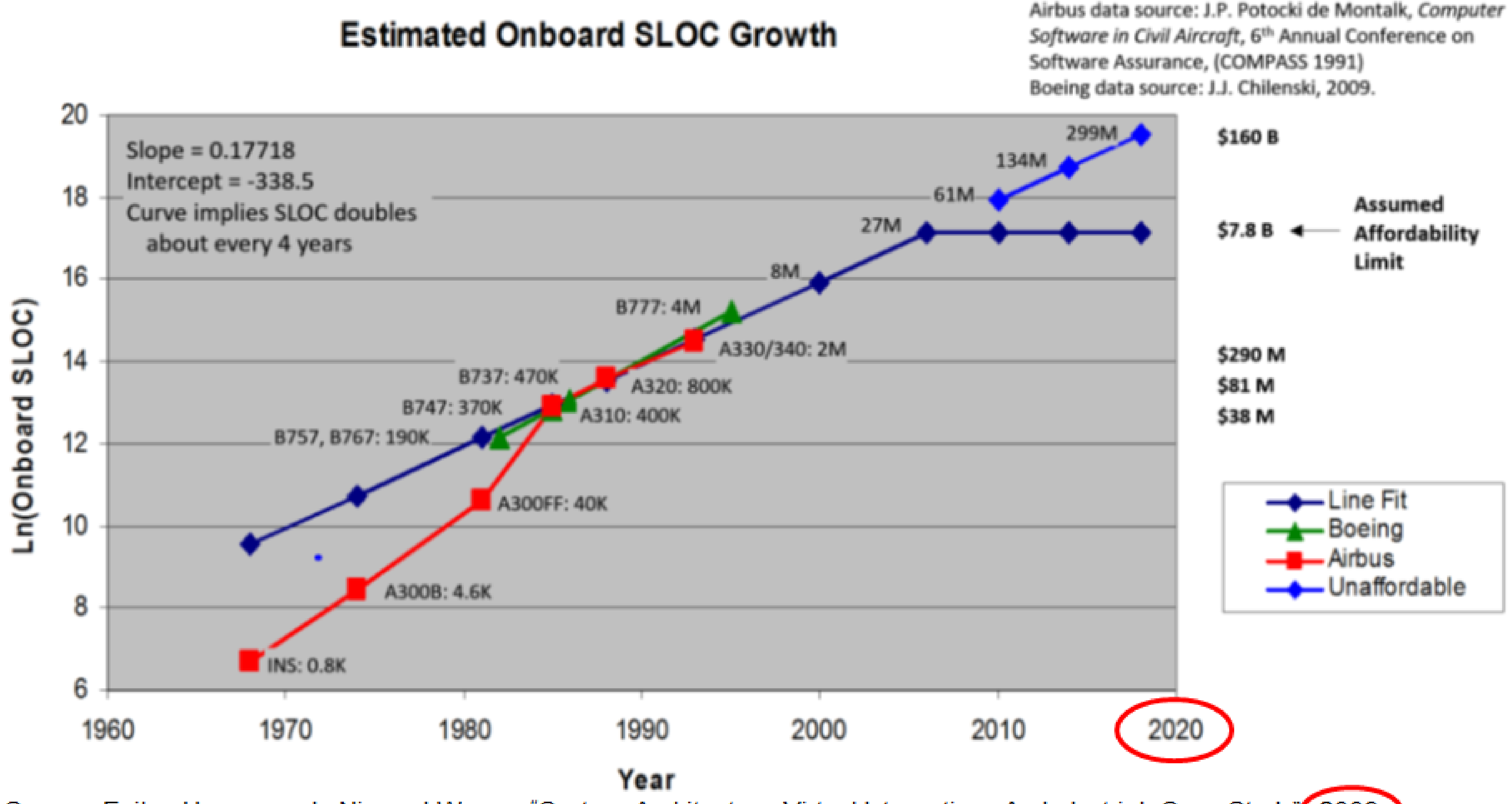
Dominique Blouin[2], Anish Bhobe[3]

[1]University of Antwerp – Flanders Make, Belgium

[2]Telecom Paris, France

[3]Institut Polytechnique de Paris, France

Increasing Systems Complexity



Source: Feiler, Hansson, de Niz and Wrage. "System Architecture Virtual Integration: An Industrial Case Study", 2009.

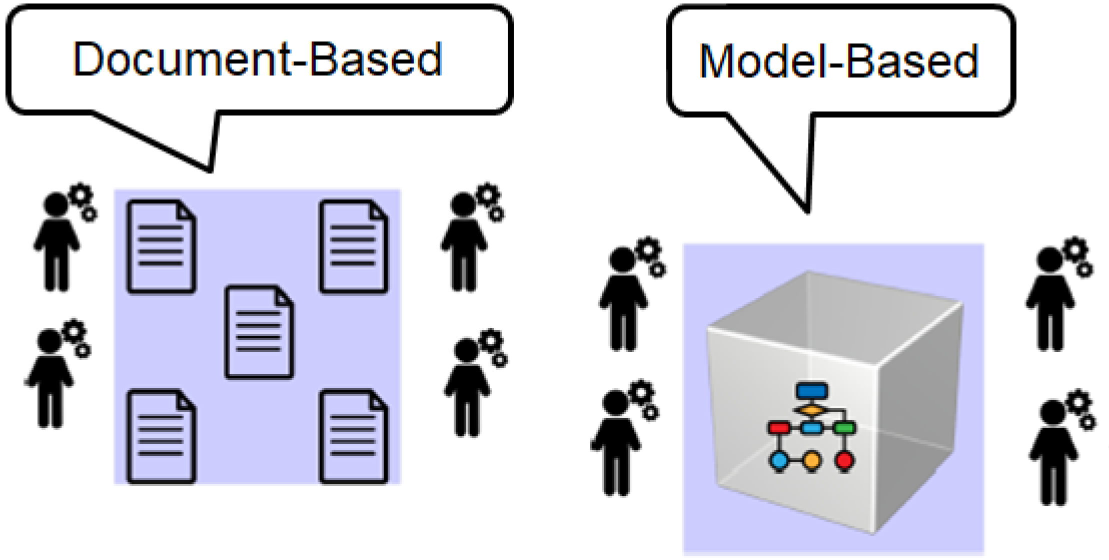
Non-Linear Development Effort Increase

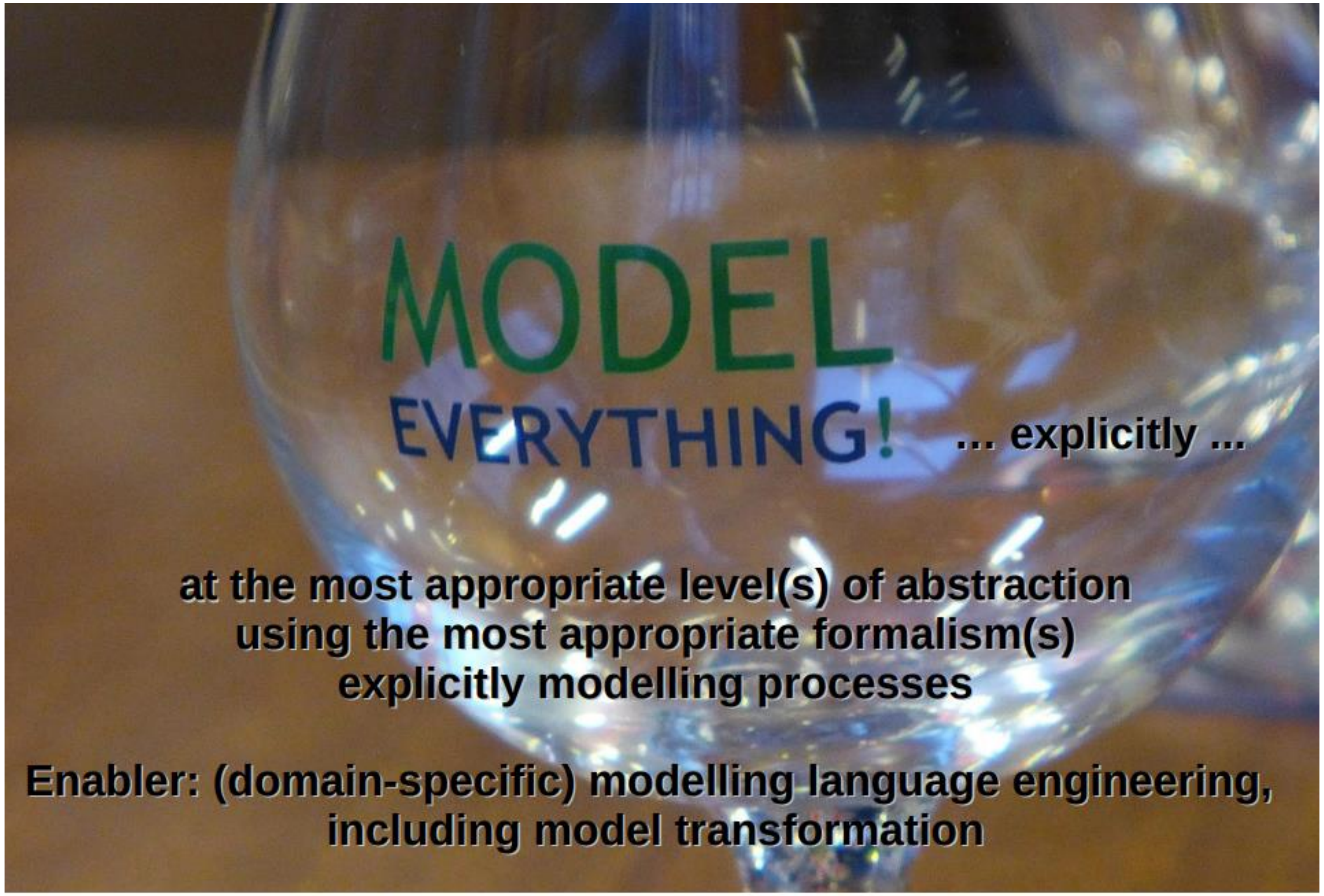
- **F35** SLOC / **F16** SLOC ~ 175
- **F35** Effort / **F16** Effort ~ **300**
  - Source: SAVI Project (https://savi.avsi.aero/)

- A400M:
  - Over 10 years delayed.
  - 6.2 billion euros over budget (30% overrun).
  - Source: https://www.rt.com/business/airbus-a400m-france-delays-561/

Paradigm Shift: Model-Based Systems Engineering (MBSE)

- From natural language documents to **models**.
- Provide common **vocabulary**.
- Enforce more **precision**.
- Allow building **tools** to process specifications (models).
- Allow detecting errors / inconsistencies **early** with these tools.
- Quite **effective** for avionics development (> 25 % costs reduction).





... explicitly ...

at the most appropriate level(s) of abstraction
using the most appropriate formalism(s)
explicitly modelling processes

Enabler: (domain-specific) modelling language engineering, including model transformation

Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. Simulation: Transactions of the Society for Modeling and Simulation International , 80(9):433- 450, September 2004. Special Issue: Grand Challenges for Modeling and Simulation.

Multi-Paradigm Modeling for Cyber-Physical Systems

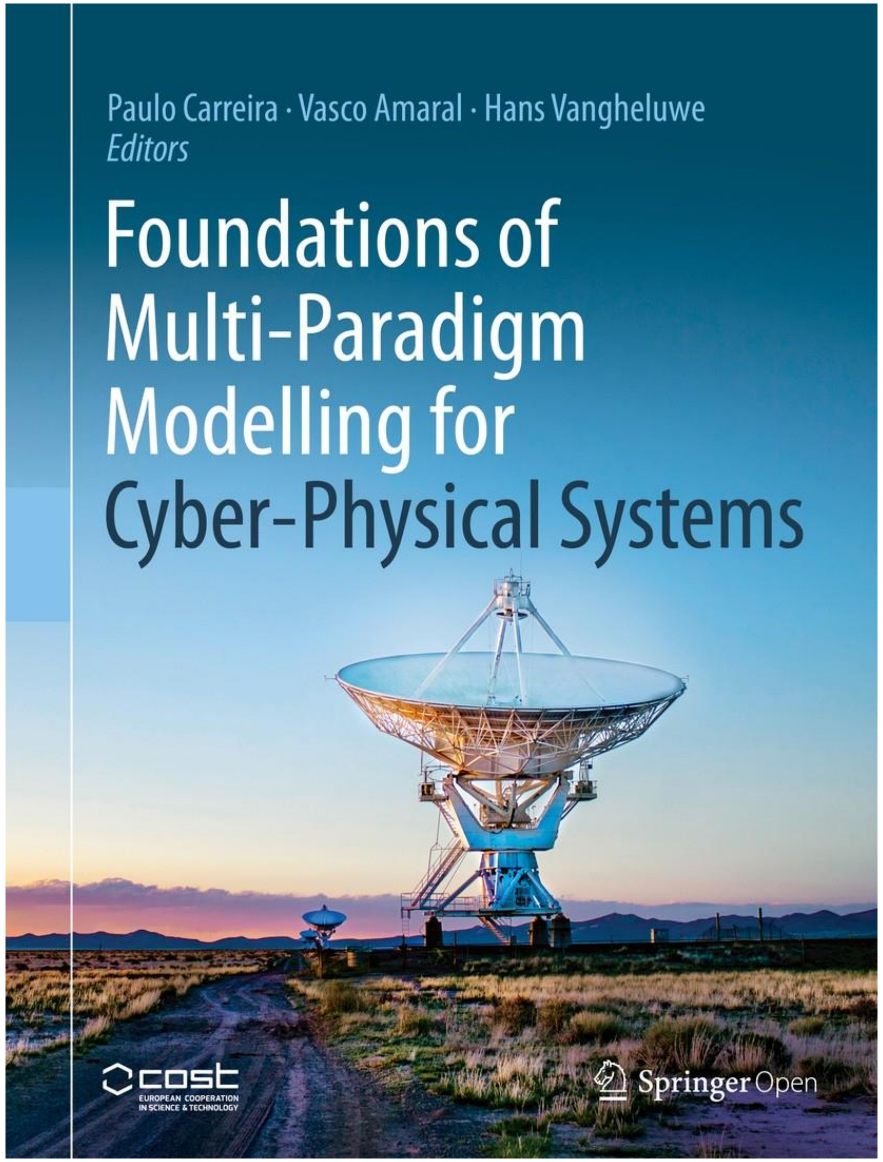
MPM4CPS

cost
EUROPEAN COOPERATION IN SCIENCE AND TECHNOLOGY

mpm4cps.eu

Hans is the pope

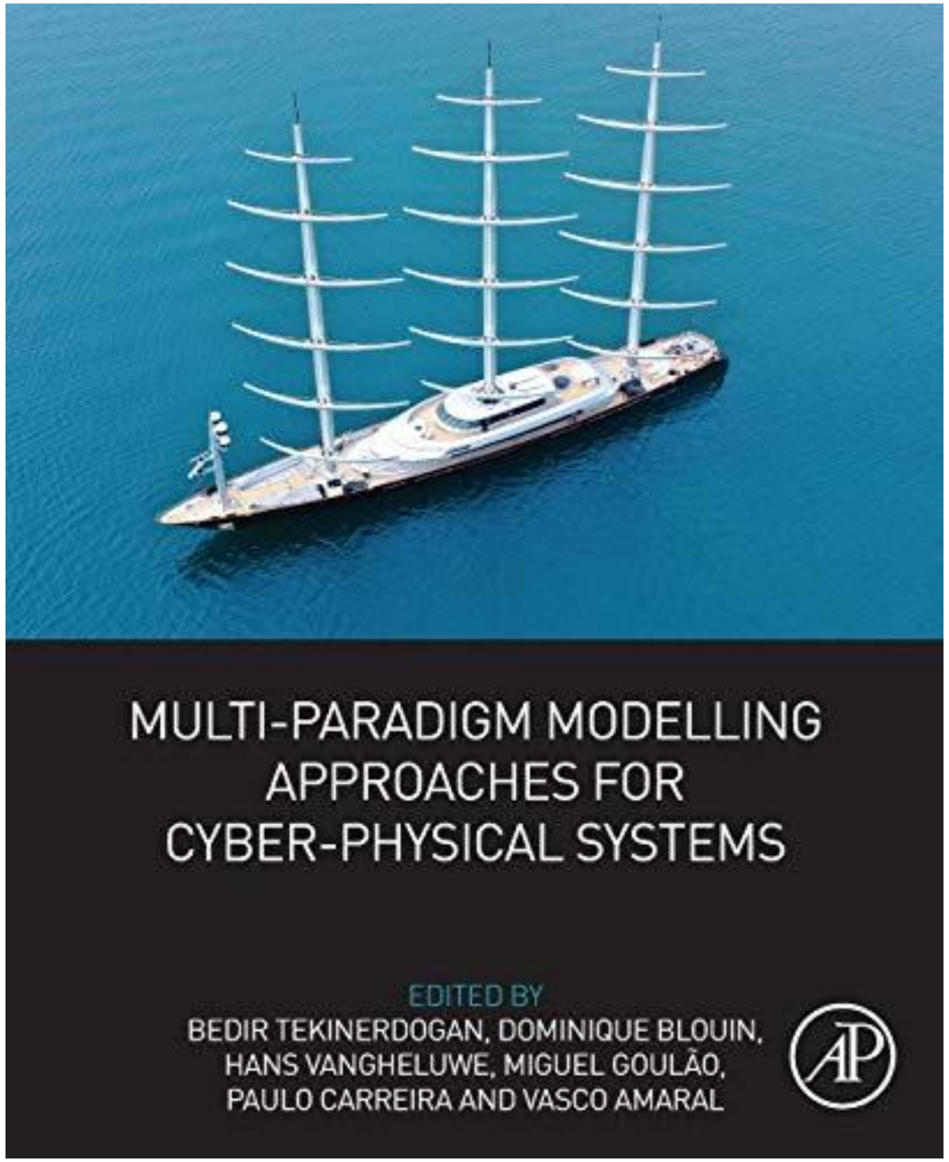
and Dominique is the bishop!


Paulo Carreira · Vasco Amaral · Hans Vangheluwe
Editors

Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems

cost EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY

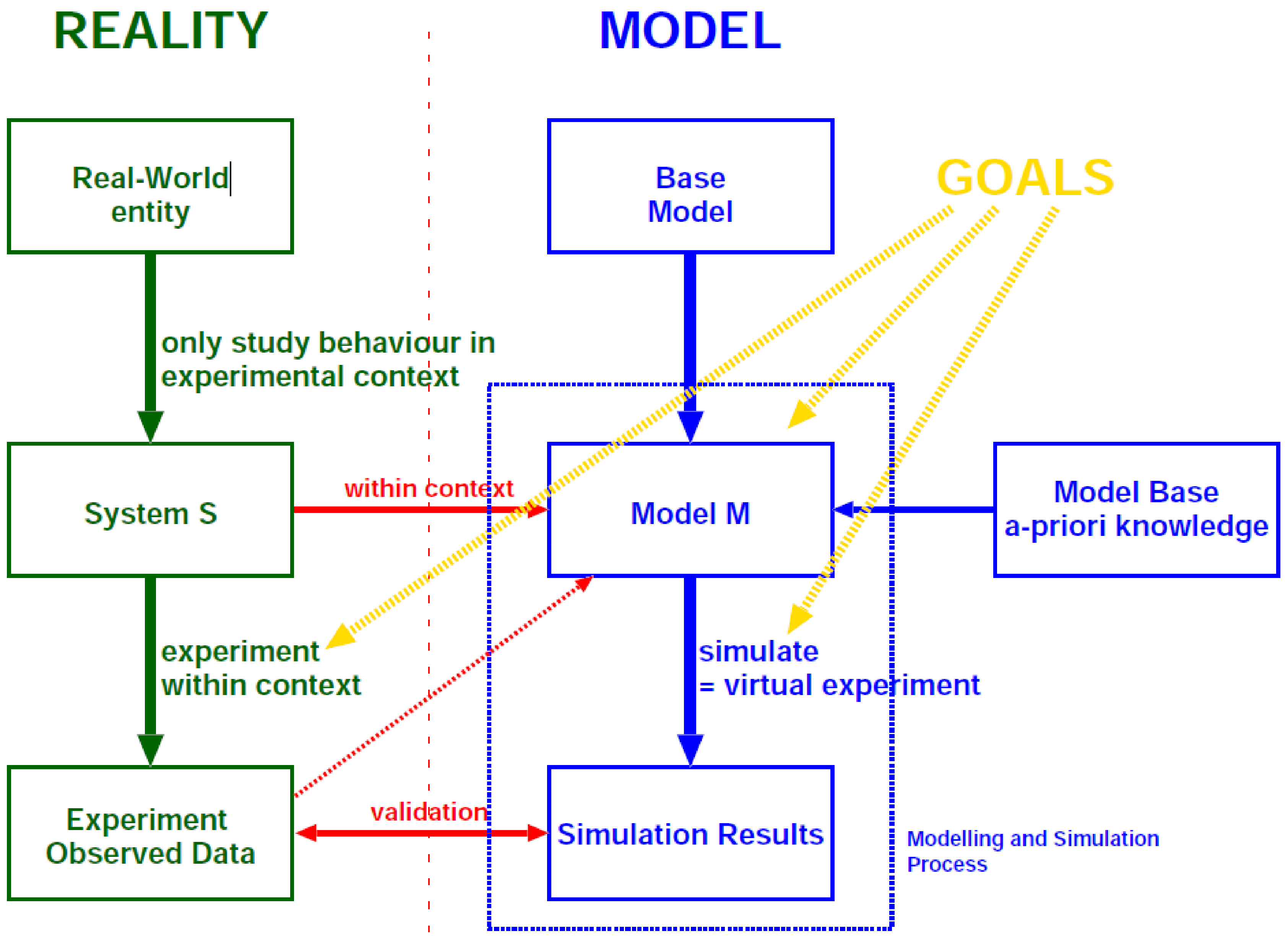
Springer Open


MULTI-PARADIGM MODELLING APPROACHES FOR CYBER-PHYSICAL SYSTEMS

EDITED BY
BEDIR TEKINERDOGAN, DOMINIQUE BLOUIN, HANS VANGHELUWE, MIGUEL GOULÃO, PAULO CARREIRA AND VASCO AMARAL

TELECOM Paris

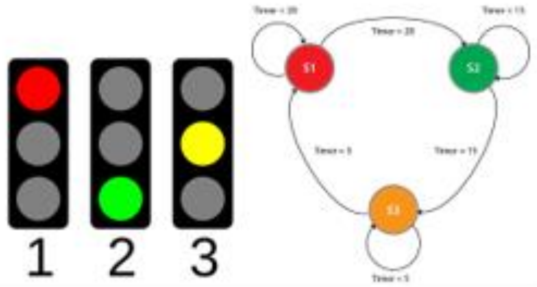
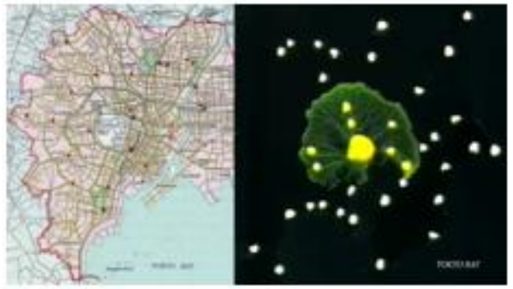
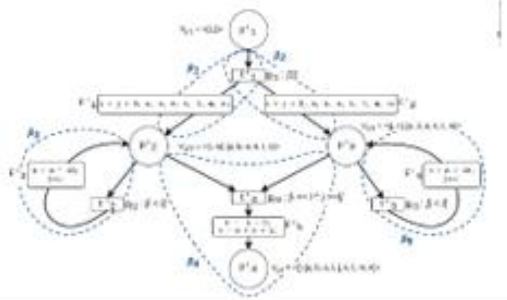
IP PARIS


REALITY
MODEL
Real-World entity
only study behaviour in experimental context
System S
experiment within context
Experiment Observed Data
Base Model
GOALS
within context
Model M
Model Base a-priori knowledge
simulate = virtual experiment
validation
Simulation Results
Modelling and Simulation Process

Bernard P. Zeigler. *Multi-faceted Modelling and Discrete-Event Simulation.* Academic Press, 1984.

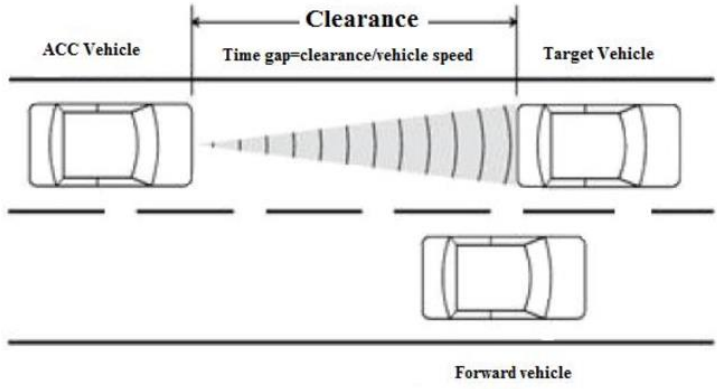
disclaimer

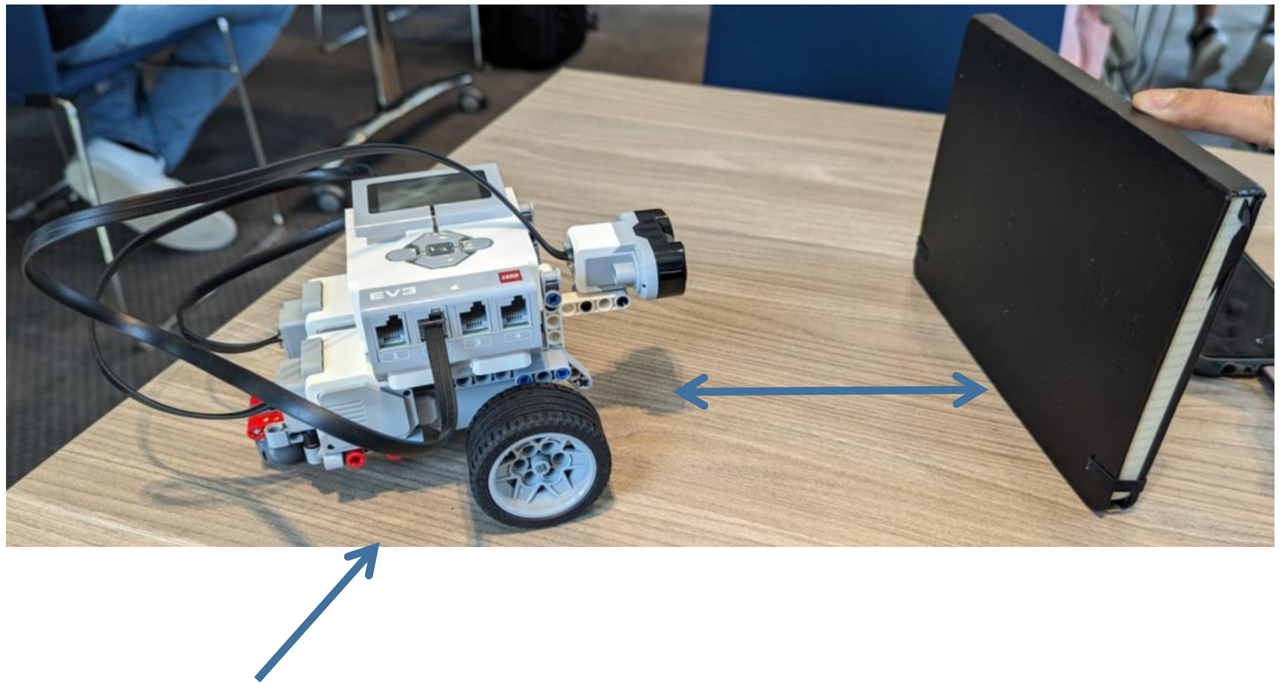
- The model need not always be 'conceptual', and the modelled system need not always be 'real'

	Real Model	Conceptual Model
Real System		
Conceptual System		

Case-Study

Adaptive Cruise Control System





The actual robot that you are going to use.

2 hands-on tutorials with foundations in Multi-Paradigm Modeling

Case Study: Adaptive Cruise Control System (ACCS)

1a: Modeling the ACCS using OpenModelica

Rakshit Mittal[1], Hans Vangheluwe[1]

1b: Verifying ACCS Safety Requirements using UPPAAL

Rizwan Parveen[2]

2a: Modeling and Analyzing the Architecture of the ACCS controller using AADL

Dominique Blouin[2], Anish Bhobe[3]

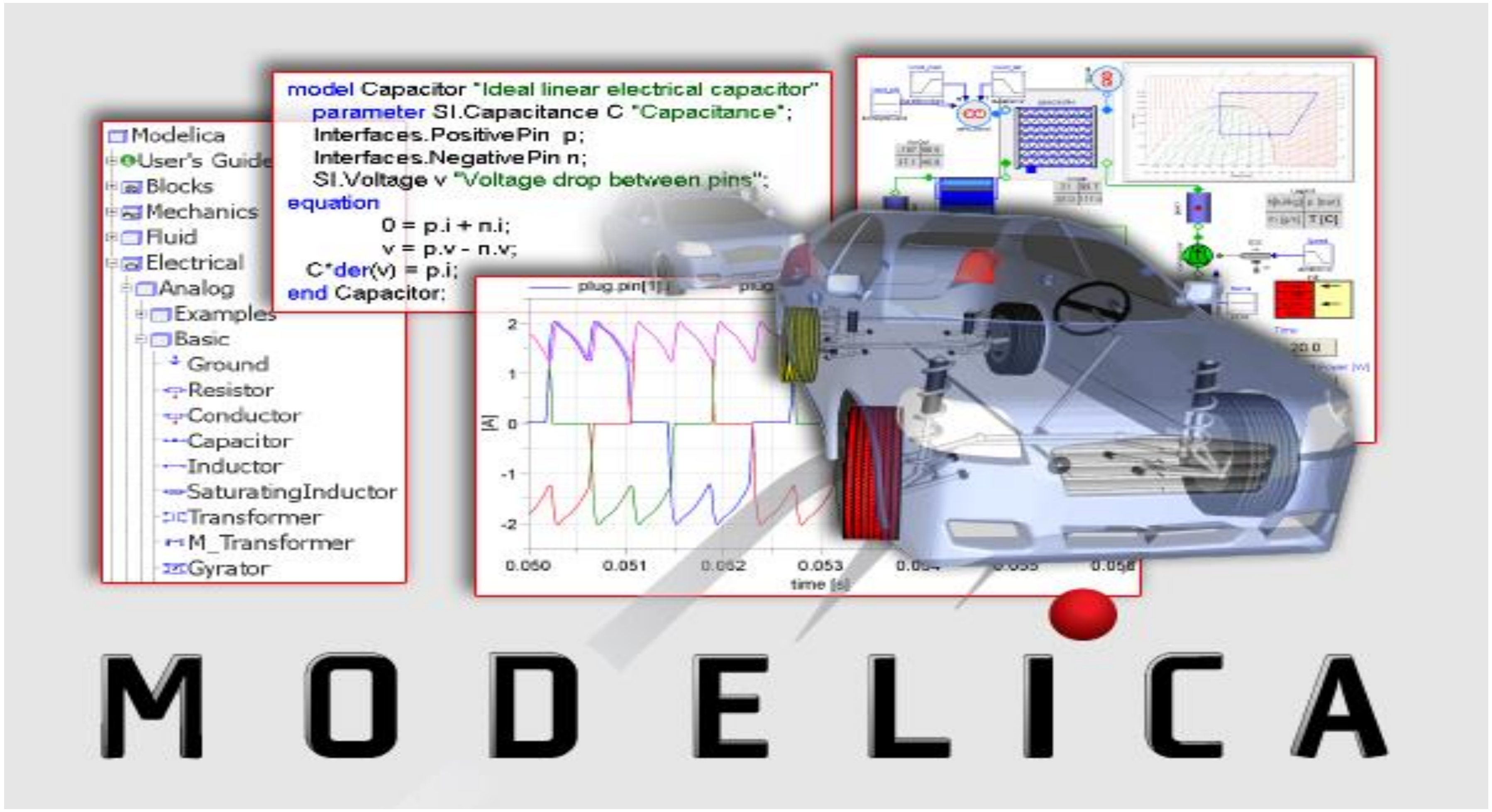
2b: Synthesizing Code for the ACCS controller using RAMSES

Dominique Blouin[2], Anish Bhobe[3]

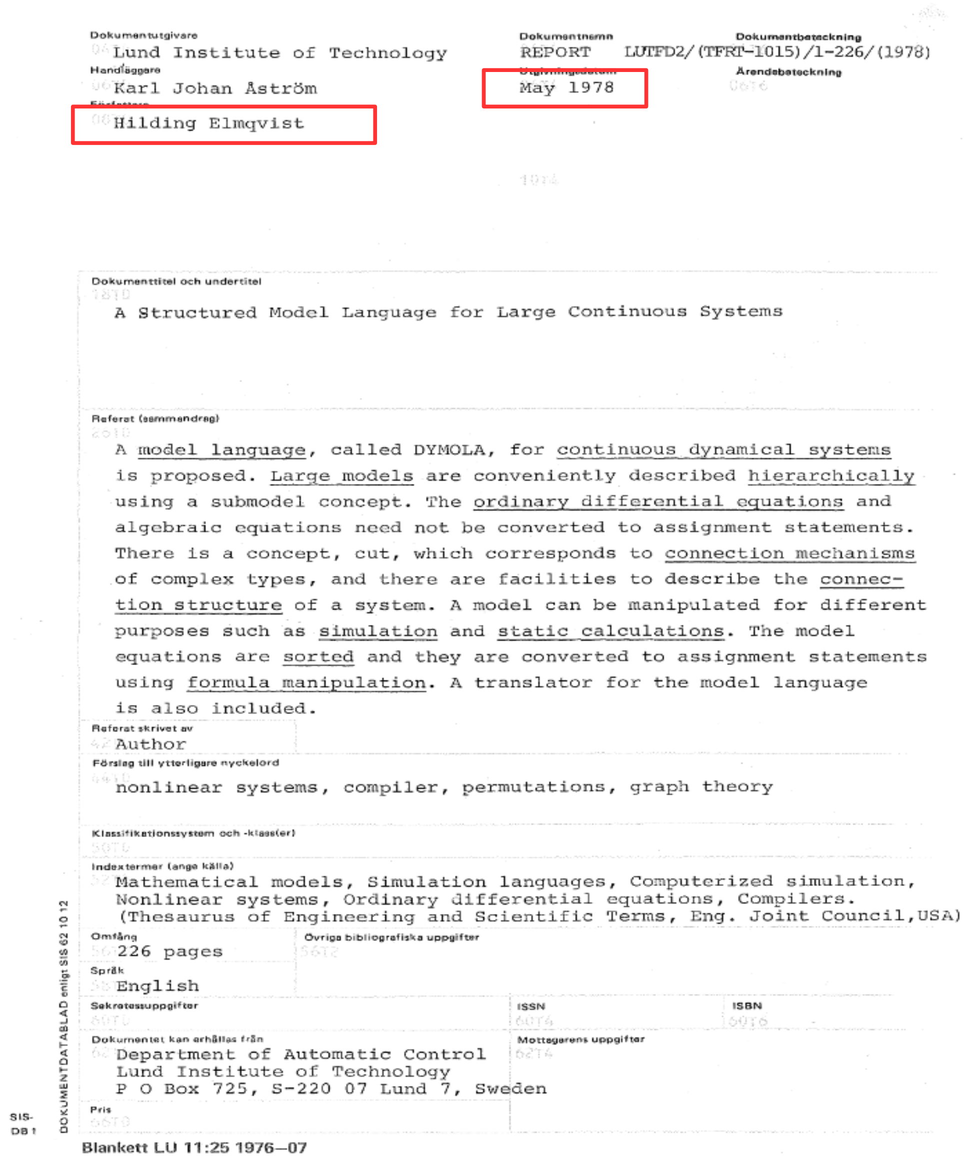
[1]University of Antwerp – Flanders Make, Belgium

[2]Telecom Paris, France

[3]Institut Polytechnique de Paris, France



```
model Capacitor "Ideal linear electrical capacitor"
  parameter SI.Capacitance C "Capacitance";
  Interfaces.PositivePin p;
  Interfaces.NegativePin n;
  SI.Voltage v "Voltage drop between pins";
equation
         0 = p.i + n.i;
         v = p.v - n.v;
  C*der(v) = p.i;
end Capacitor;
```



Dokumentutgivare	Lund Institute of Technology
Handläggare	Karl Johan Åström
Författare	Hilding Elmqvist
Dokumentnamn	REPORT
Dokumentbeteckning	LUTFD2/(TFRT-1015)/1-226/(1978)
Utgivningsdatum	May 1978
Ärendebeteckning	

Dokumenttitel och undertitel

A Structured Model Language for Large Continuous Systems

Referat (sammandrag)

A model language, called DYMOLA, for continuous dynamical systems is proposed. Large models are conveniently described hierarchically using a submodel concept. The ordinary differential equations and algebraic equations need not be converted to assignment statements. There is a concept, cut, which corresponds to connection mechanisms of complex types, and there are facilities to describe the connection structure of a system. A model can be manipulated for different purposes such as simulation and static calculations. The model equations are sorted and they are converted to assignment statements using formula manipulation. A translator for the model language is also included.

Referat skrivet av

Author

Förslag till ytterligare nyckelord

nonlinear systems, compiler, permutations, graph theory

Klassifikationssystem och -klass(er)

Indextermer (ange källa)

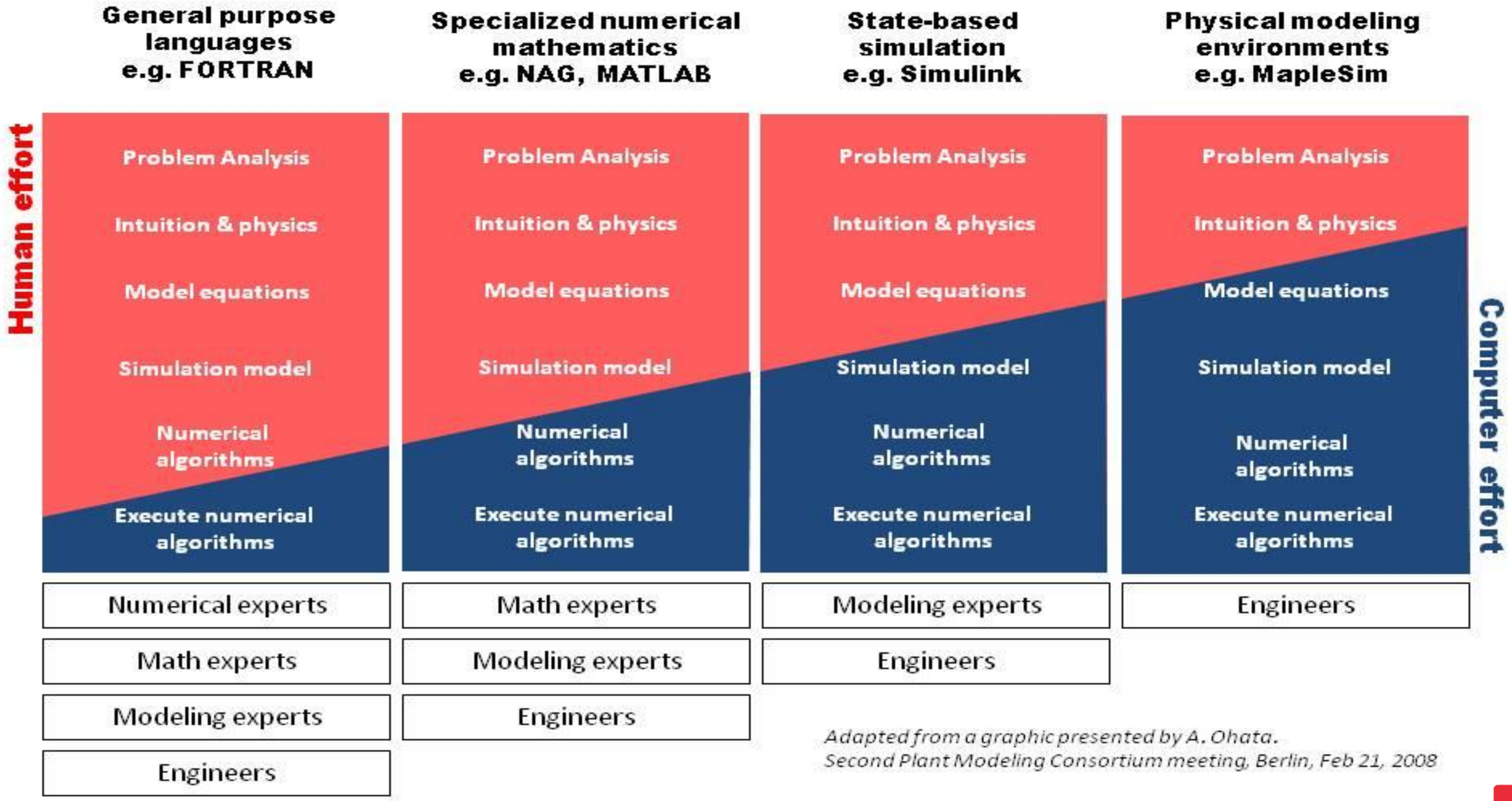
Mathematical models, Simulation languages, Computerized simulation, Nonlinear systems, Ordinary differential equations, Compilers. (Thesaurus of Engineering and Scientific Terms, Eng. Joint Council, USA)

Omfång	226 pages
Övriga bibliografiska uppgifter	
Språk	English
Sekretessuppgifter	
ISSN	
ISBN	
Dokumentet kan erhållas från	Department of Automatic Control Lund Institute of Technology P O Box 725, S-220 07 Lund 7, Sweden
Mottagarens uppgifter	
Pris	

SIS-DB 1

DOKUMENTDATABLAD enligt SIS 62 10 12

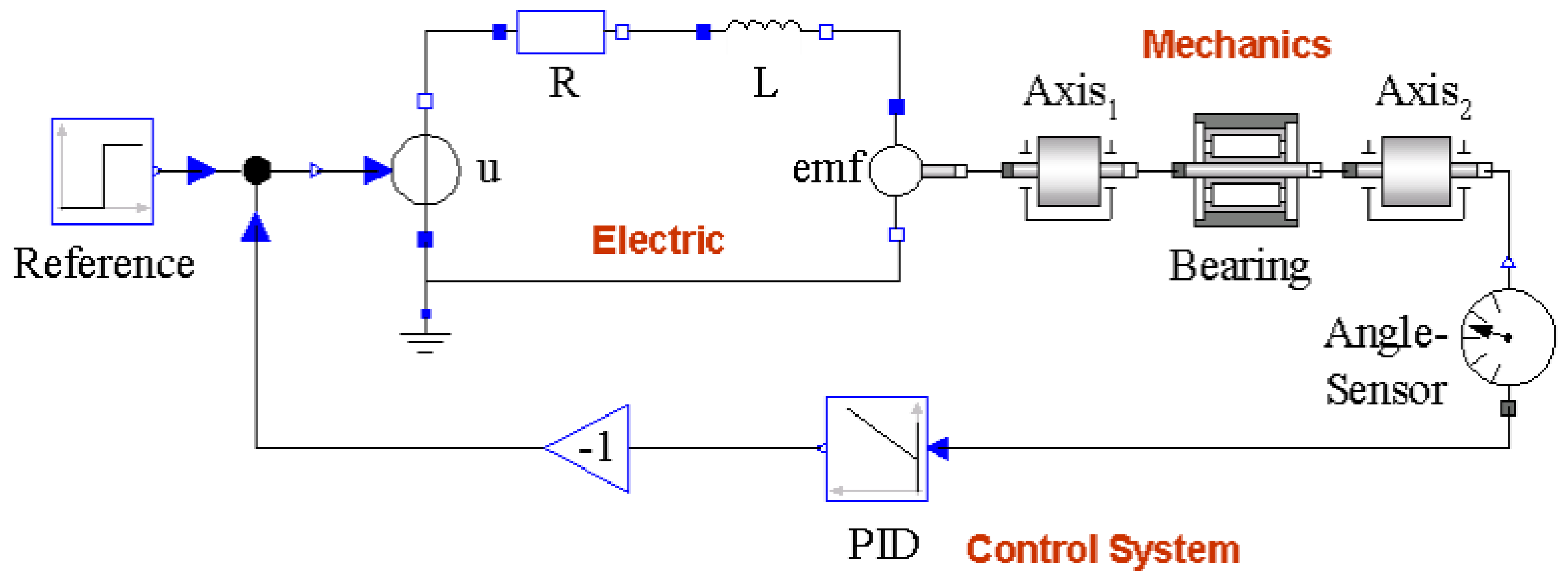
Blankett LU 11:25 1976–07



Adapted from a graphic presented by A. Ohata.
Second Plant Modeling Consortium meeting, Berlin, Feb 21, 2008

Multi-Domain Modeling

http://www.modelica.org

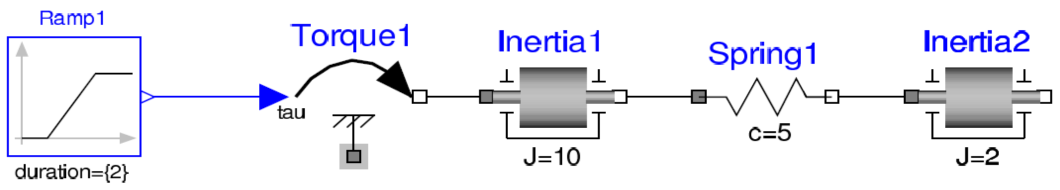


this slide from Peter Fritzson's Modelica tutorial

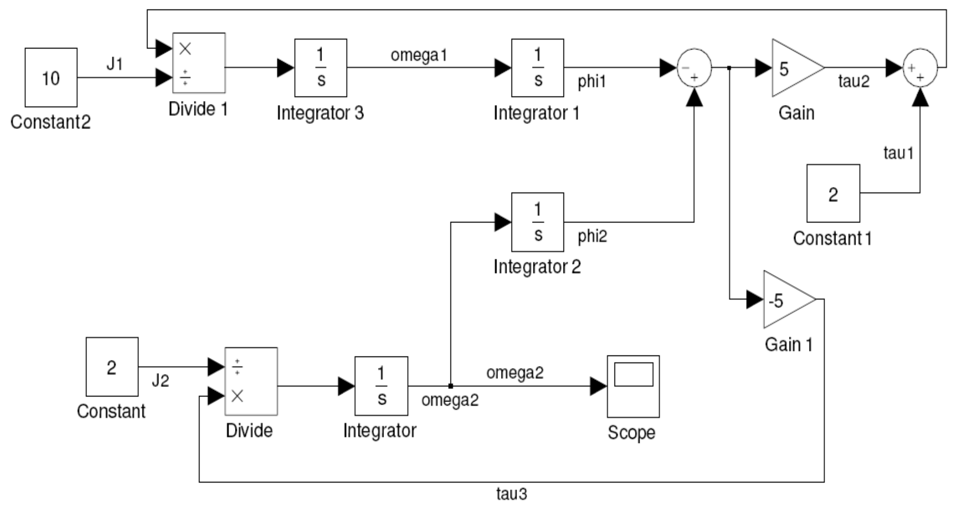
Visual Acausal Hierarchical Component Modeling

Keeps the physical structure

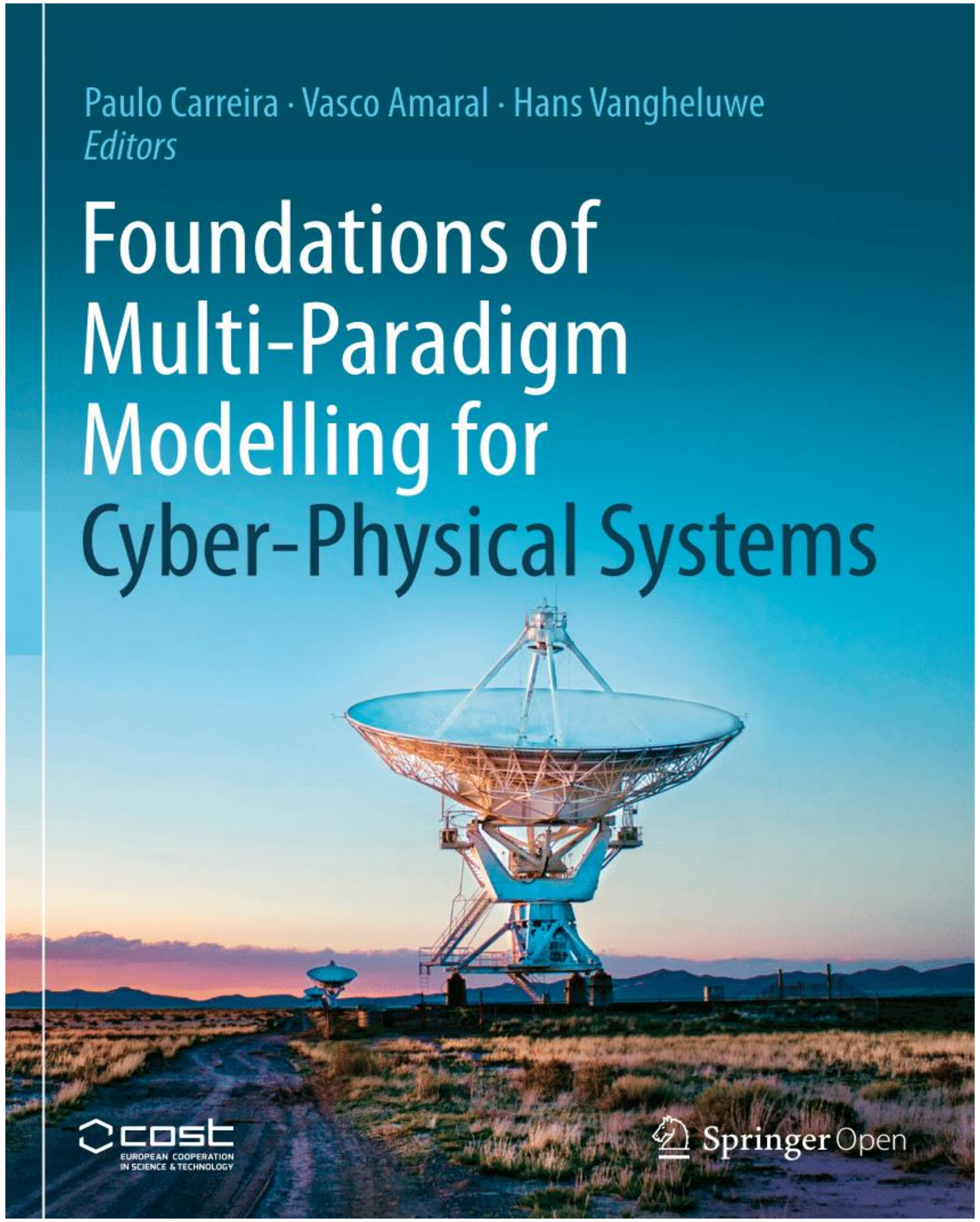
Acausal model (Modelica)



Causal block-based model (Simulink)



this slide from Peter Fritzson's Modelica tutorial



Paulo Carreira · Vasco Amaral · Hans Vangheluwe
Editors

Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems

cost EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY

Springer Open

Modelica

https://modelica.org/documents/ModelicaTutorial14.pdf

OpenModelica

https://openmodelica.org/

Modelica by Example

by Dr. Michael M. Tiller

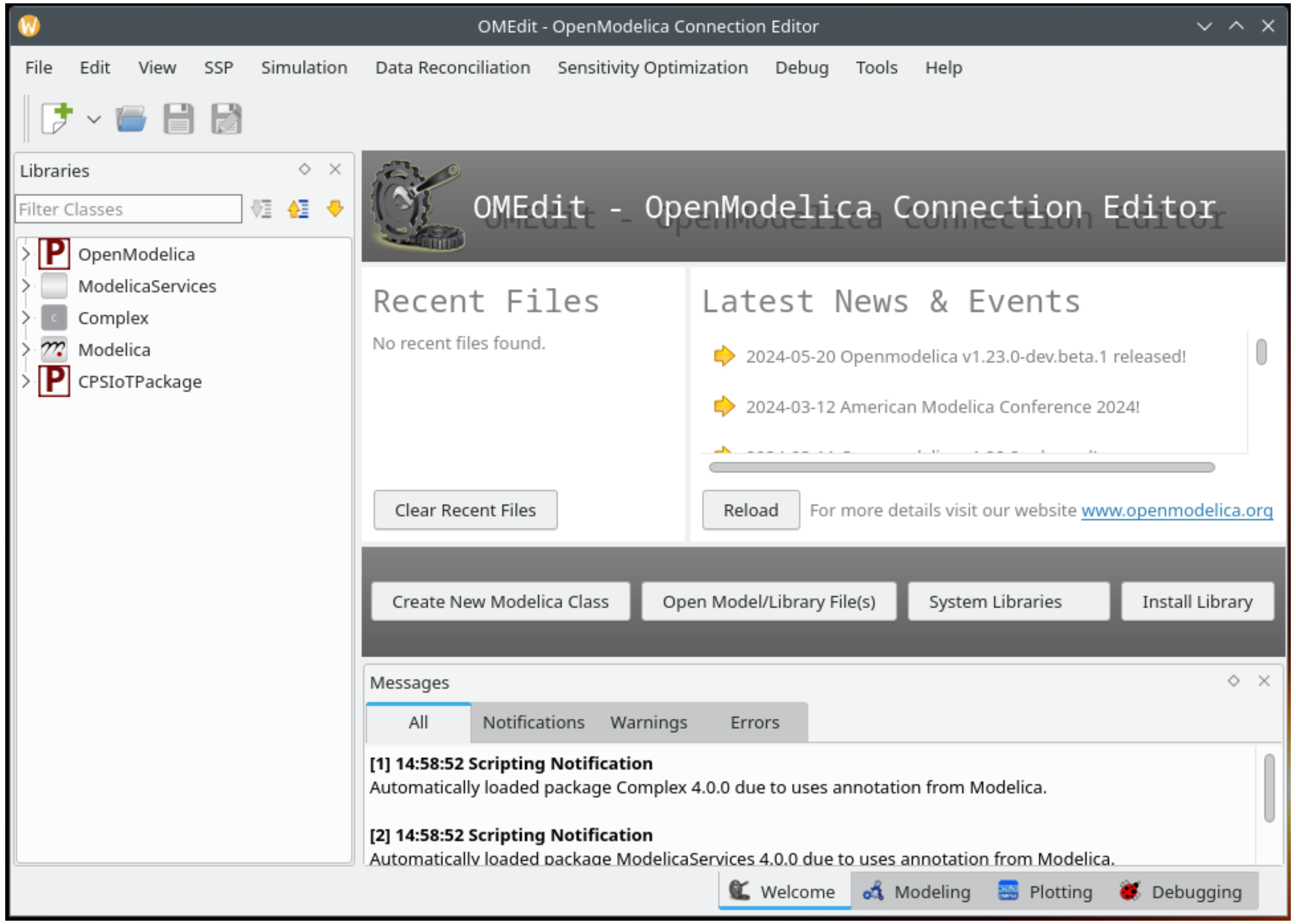
https://mbe.modelica.university/

Fritzson P. (2020) Modelica: Equation-Based, Object-Oriented Modelling of Physical Systems.
In: Carreira P., Amaral V., Vangheluwe H. (eds) Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems. Springer, Cham.
https://doi.org/10.1007/978-3-030-43946-0_3

TELECOM Paris

IP PARIS

The tool:
OpenModelica Connection Editor (OMEdit)



[1] 14:58:52 Scripting Notification
Automatically loaded package Complex 4.0.0 due to uses annotation from Modelica.

[2] 14:58:52 Scripting Notification
Automatically loaded package ModelicaServices 4.0.0 due to uses annotation from Modelica.

Download the tool from:
https://openmodelica.org/#

The resources:
download from
https://nextcloud.rakshitmittal.net/s/iY4qRkgkW9yx8WB
or request a pen-drive!

Equation-Based Object-Oriented Modelling of the Physics, with Modelica

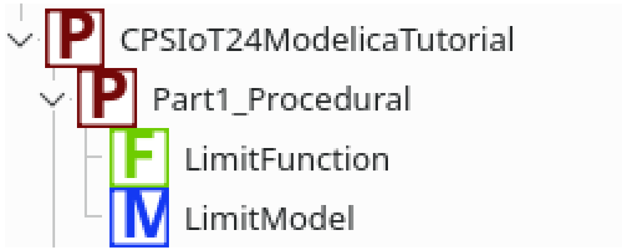
- Programming: procedural code (function/algorithm)
- Equation-based (a-causal) modelling
- Behind the scenes: numerical approximations
- Object-Oriented modelling
- Libraries and the MSL
- Controller Modelling
- Extra time: Hiding IP: Composition of Functional Mockup Units (FMI)

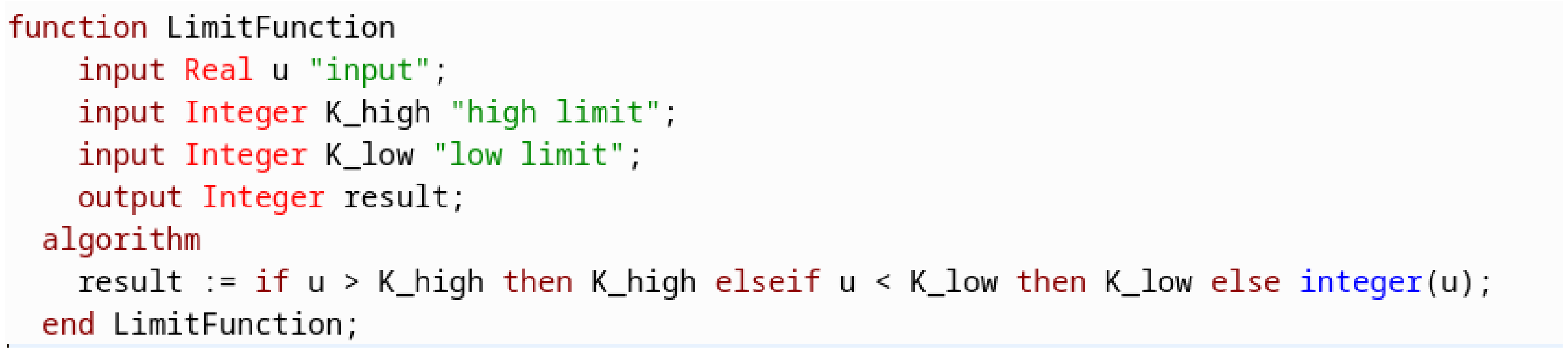
Equation-Based Object-Oriented Modelling of the Physics, with Modelica

- Programming: procedural code (function/algorithm)
- Equation-based (a-causal) modelling
- Behind the scenes: numerical approximations
- Object-Oriented modelling
- Libraries and the MSL
- Controller Modelling
- Extra time: Hiding IP: Composition of Functional Mockup Units (FMI)

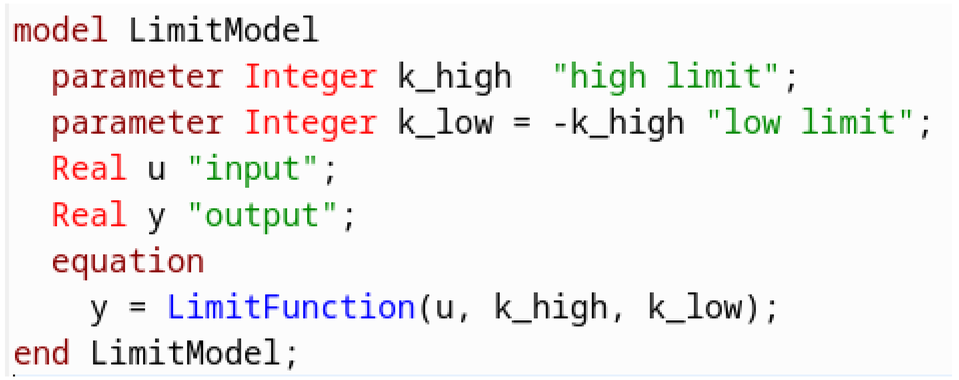
5 mins

The motor should not move too fast!
So the input to the motor controller is limited to [-300, 300].
Simulate the function using the test-bed. Modify the parameters and observe simulation output.





```
function LimitFunction
    input Real u "input";
    input Integer K_high "high limit";
    input Integer K_low "low limit";
    output Integer result;
  algorithm
    result := if u > K_high then K_high elseif u < K_low then K_low else integer(u);
  end LimitFunction;
```



```
model LimitModel
  parameter Integer k_high  "high limit";
  parameter Integer k_low = -k_high "low limit";
  Real u "input";
  Real y "output";
  equation
    y = LimitFunction(u, k_high, k_low);
end LimitModel;
```

Equation-Based Object-Oriented Modelling of the Physics, with Modelica

- Programming: procedural code (function/algorithm)
- Equation-based (a-causal) modelling
- Behind the scenes: numerical approximations
- Object-Oriented modelling
- Libraries and the MSL
- Controller Modelling
- Extra time: Hiding IP: Composition of Functional Mockup Units (FMI)

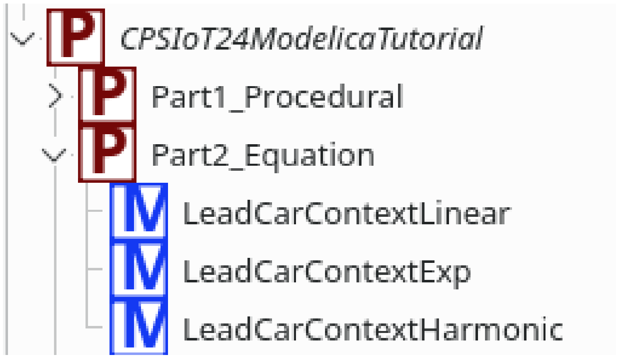
10 mins

The position of the lead car can be described by differential equations.
Three different kinds are already provided.
Simulate them, and then also create your own custommodel!

```
model LeadCarContextLinear
  Real x(start = 10);
  equation
    der(x) = 5;
end LeadCarContextLinear;
```

```
model LeadCarContextExp
  Real x(start = 10);
  equation
    der(x) = x;
end LeadCarContextExp;
```

```
model LeadCarContextHarmonic
  Real x(start = 10);
  Real v(start = 0);
  equation
    der(x) = v;
    der(v) = -x;
// x(t) = A*sin(t) + B*cos(t)
// v(t) = A*cost(t) - B*sin(t)
end LeadCarContextHarmonic;
```



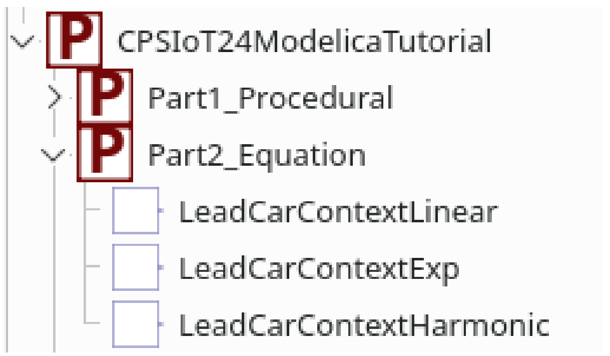
```
CPSIoT24ModelicaTutorial
  Part1_Procedural
  Part2_Equation
    LeadCarContextLinear
    LeadCarContextExp
    LeadCarContextHarmonic
```

Equation-Based Object-Oriented Modelling of the Physics, with Modelica

- Programming: procedural code (function/algorithm)
- Equation-based (a-causal) modelling
- Behind the scenes: numerical approximations
- Object-Oriented modelling
- Libraries and the MSL
- Controller Modelling
- Extra time: Hiding IP: Composition of Functional Mockup Units (FMI)

10 mins

Simulate the harmonic equation with different settings:


CPSIoT24ModelicaTutorial
Part1_Procedural
Part2_Equation
LeadCarContextLinear
LeadCarContextExp
LeadCarContextHarmonic

<u>Simulation 1</u>

solver :	dassl
stop-time:	20 s
step-size :	0.02 s

<u>Simulation 2</u>

solver :	euler
stop-time:	20 s
step-size :	0.5 s

```
model LeadCarContextHarmonic
  Real x(start = 10);
  Real v(start = 0);
  equation
    der(x) = v;
    der(v) = -x;
// x(t) = A*sin(t) + B*cos(t)
// v(t) = A*cost(t) - B*sin(t)
end LeadCarContextHarmonic;
```

Which simulation is correct?

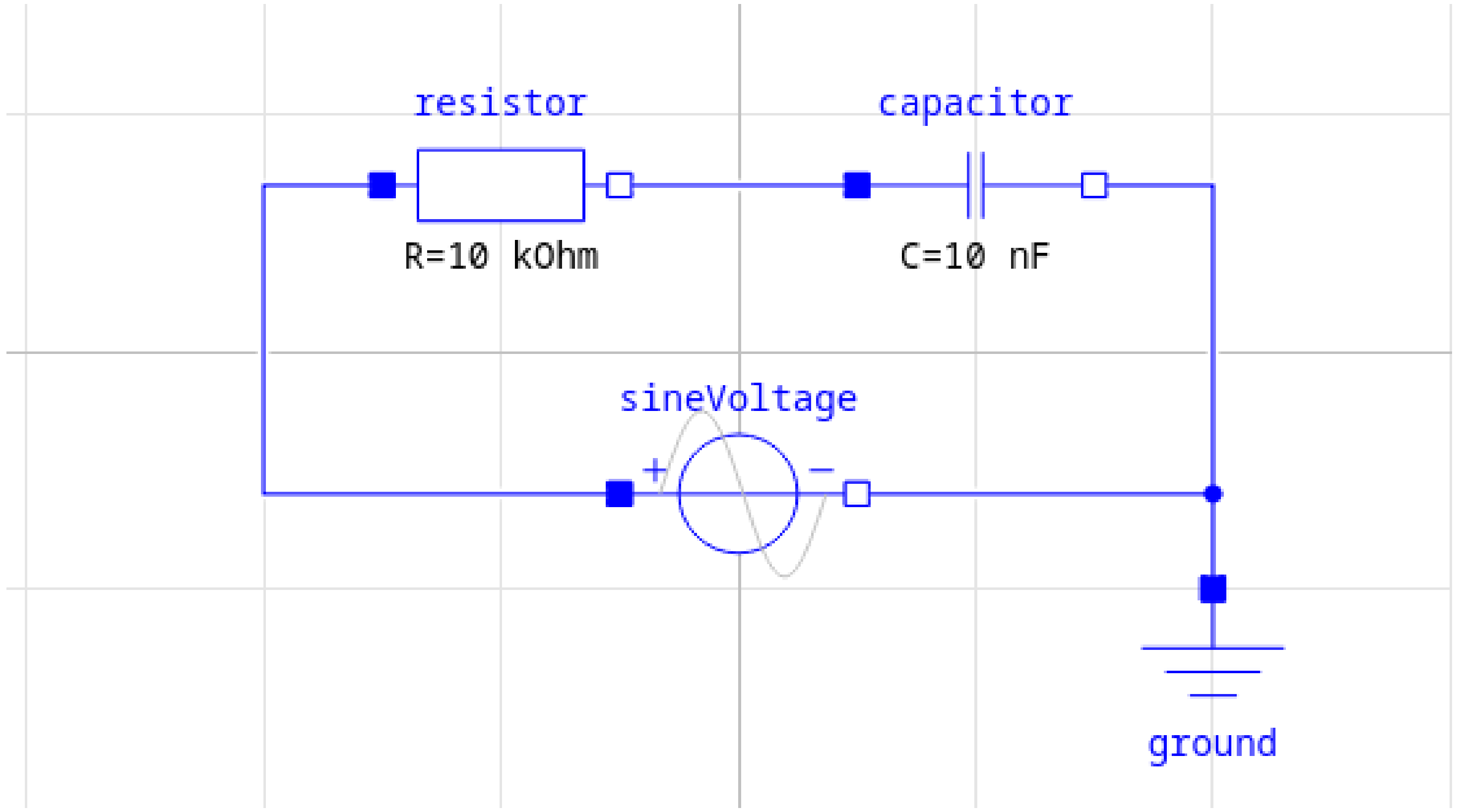
Notice the numerical in/stability.
Stability => The parametric plot should be bounded.

So, it not just about having the correct model, but also using the correct solver settings!

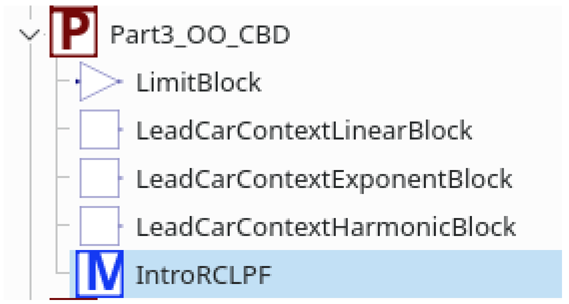
Equation-Based Object-Oriented Modelling of the Physics, with Modelica

- Programming: procedural code (function/algorithm)
- Equation-based (a-causal) modelling
- Behind the scenes: numerical approximations
- Object-Oriented modelling
- Libraries and the MSL
- Controller Modelling
- Extra time: Hiding IP: Composition of Functional Mockup Units (FMI)

Object-Orientation: concepts like classes/types, instances, encapsulation, specialization



An exemplar low-pass RC circuit



Part3_OO_CBD
LimitBlock
LeadCarContextLinearBlock
LeadCarContextExponentBlock
LeadCarContextHarmonicBlock
IntroRCLPF

Electrical Types

```
type Time = Real (final quantity="Time", final unit="s");
type ElectricPotential = Real (final quantity="ElectricPotential",
                               final unit="V");
type Voltage = ElectricPotential;
type ElectricCurrent = Real (final quantity="ElectricCurrent",
                             final unit="A");
type Current = ElectricCurrent;
```

Electrical Pin Interface

```
connector PositivePin "Positive pin of an electric component"
       Voltage v "Potential at the pin";
  flow Current i "Current flowing into the pin";
end PositivePin;
```

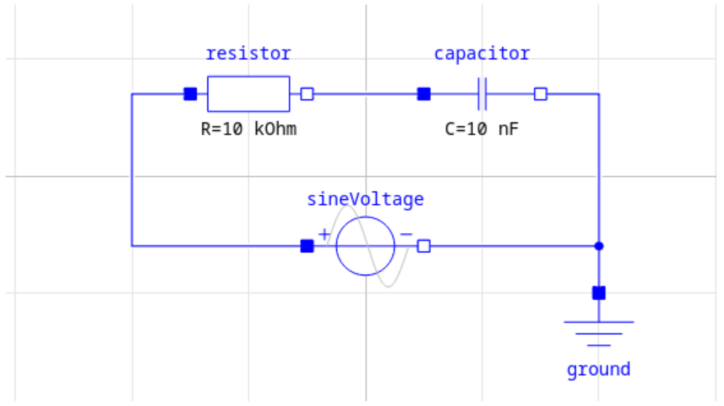
Electrical Port

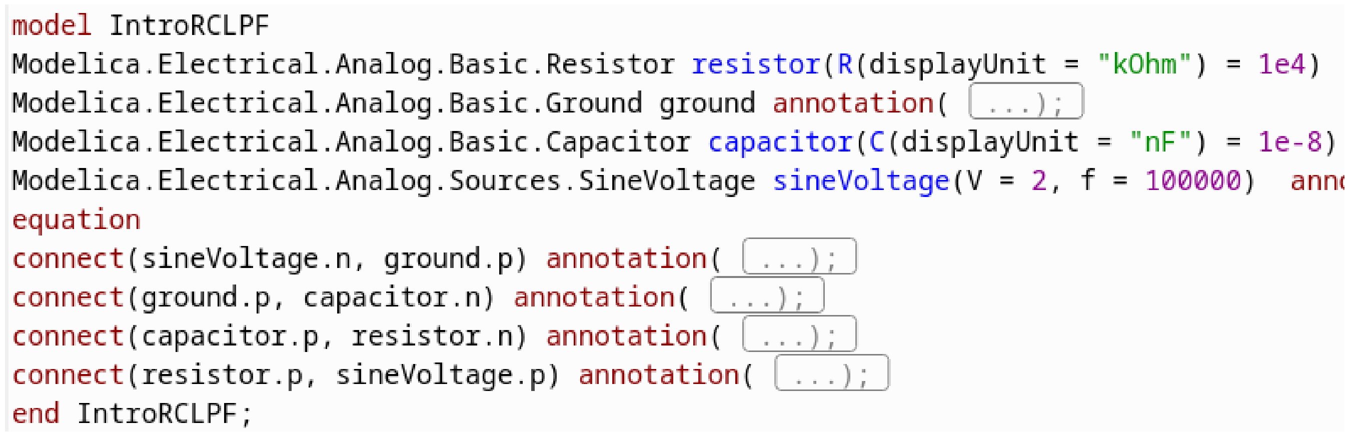
```
partial model OnePort
  "Component with two electrical pins p and n
   and current i from p to n"
  Voltage v "Voltage drop between the two pins (= p.v - n.v)";
  Current i "Current flowing from pin p to pin n";
  PositivePin p;
  NegativePin n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;
```

Electrical Resistor

```
model Resistor "Ideal linear electrical resistor"
  extends OnePort;
  parameter Resistance R=1 "Resistance";
  equation
    R*i = v;
end Resistor;
```

What is the meaning behind the connections between these re-usable blocks? How is this meaning extracted?





```
model IntroRCLPF
Modelica.Electrical.Analog.Basic.Resistor resistor(R(displayUnit = "kOhm") = 1e4)
Modelica.Electrical.Analog.Basic.Ground ground annotation( ...);
Modelica.Electrical.Analog.Basic.Capacitor capacitor(C(displayUnit = "nF") = 1e-8)
Modelica.Electrical.Analog.Sources.SineVoltage sineVoltage(V = 2, f = 100000)  ann
equation
connect(sineVoltage.n, ground.p) annotation( ...);
connect(ground.p, capacitor.n) annotation( ...);
connect(capacitor.p, resistor.n) annotation( ...);
connect(resistor.p, sineVoltage.p) annotation( ...);
end IntroRCLPF;
```

The meaning is always: a set of Differential Algebraic Equations (DAEs) !!

They are obtained by:

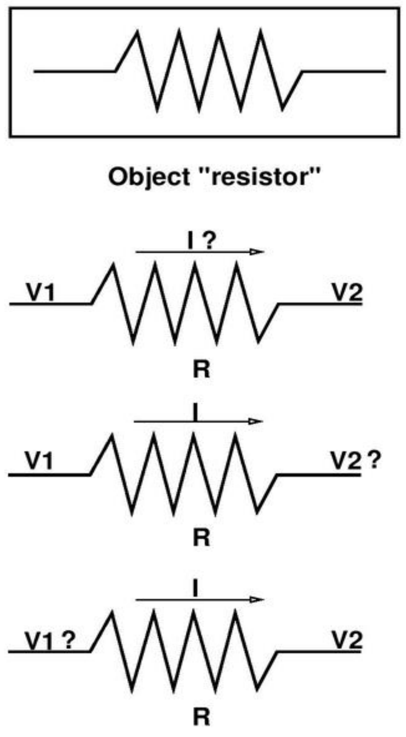
1.a. expanding inheritance

1.b. instantiation

2. flattening hierarchy, construct unique names

3. expanding connect() into equations (across vs. flow)

Object-oriented re-use and causality



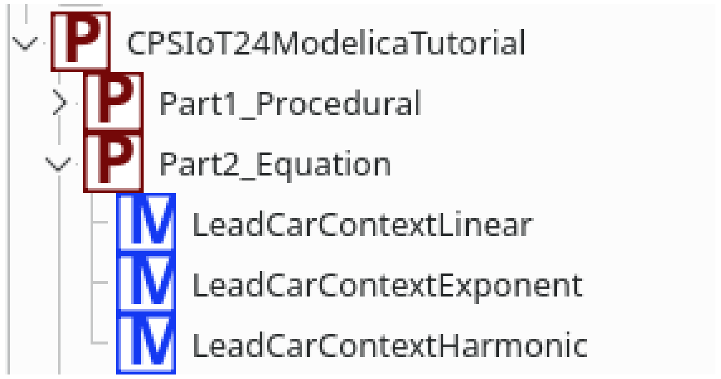
V1 – V2 = R*I

I = (V1–V2)/R

V2 = V1 – R*I

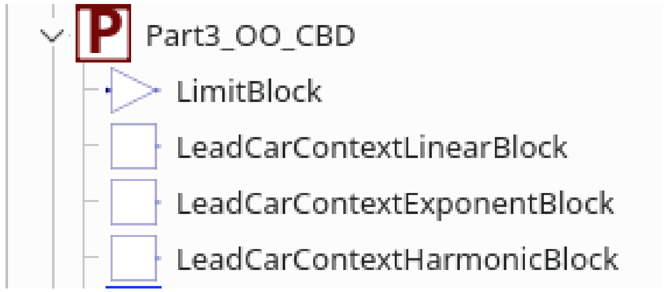
V1 = V2 + R*I

15 mins


CPSIoT24ModelicaTutorial
Part1_Procedural
Part2_Equation
LeadCarContextLinear
LeadCarContextExponent
LeadCarContextHarmonic

Recall that we created at least 4 different models.

Can we now extend those models so that they can be re-used like blocks in the Modelica graphical syntax?


Part3_OO_CBD
LimitBlock
LeadCarContextLinearBlock
LeadCarContextExponentBlock
LeadCarContextHarmonicBlock

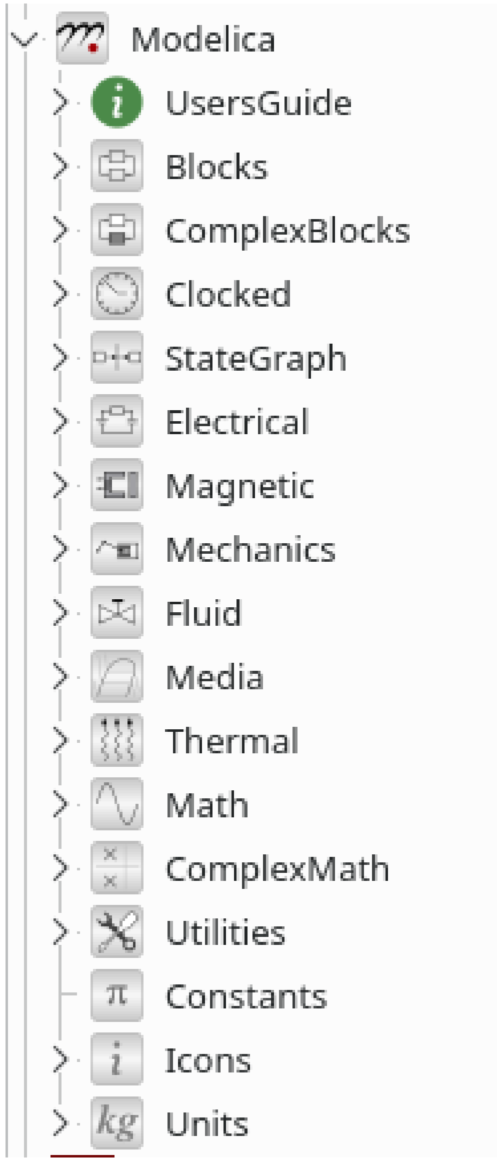
As an example, you will find (in part 3) the corresponding blocks for the four models from the previous parts of the tutorial.

You should look at the textual syntax of the models, and then use similar techniques to make the block <u>for your custom model</u>, that you created in part 2.

Equation-Based Object-Oriented Modelling of the Physics, with Modelica

- Programming: procedural code (function/algorithm)
- Equation-based (a-causal) modelling
- Behind the scenes: numerical approximations
- Object-Oriented modelling
- Libraries and the MSL
- Controller Modelling
- Extra time: Hiding IP: Composition of Functional Mockup Units (FMI)

MSL - Modelica Standard Library

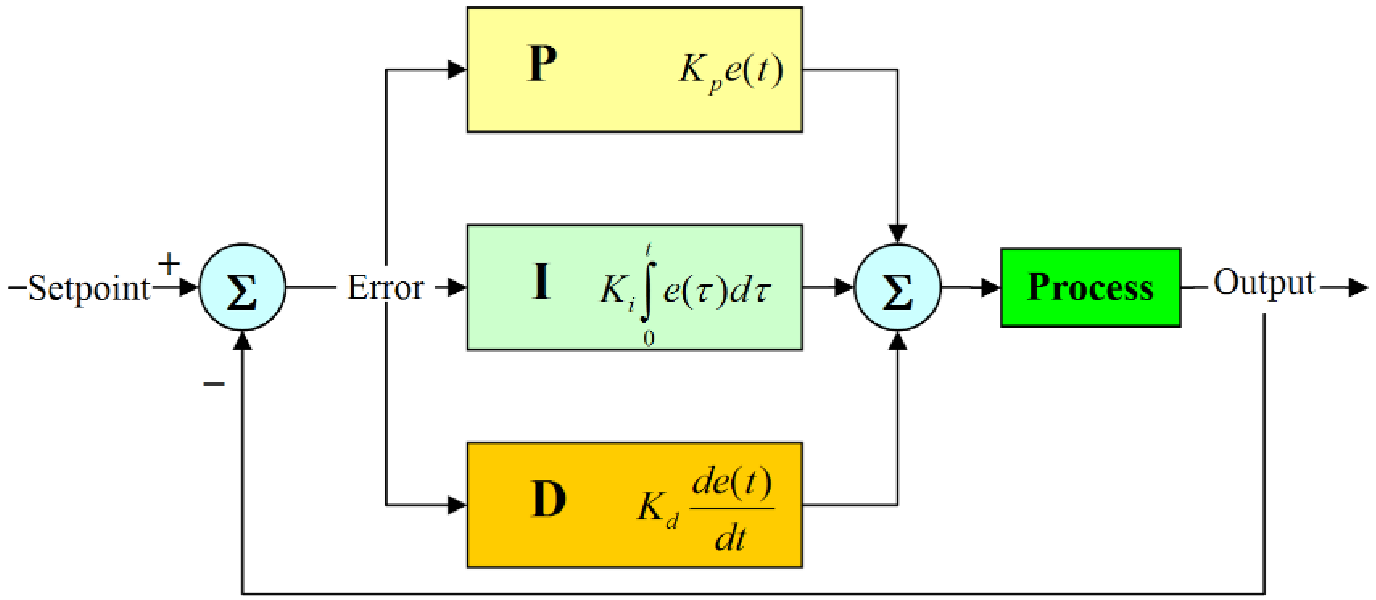


- Modelica
  - UsersGuide
  - Blocks
  - ComplexBlocks
  - Clocked
  - StateGraph
  - Electrical
  - Magnetic
  - Mechanics
  - Fluid
  - Media
  - Thermal
  - Math
  - ComplexMath
  - Utilities
  - Constants
  - Icons
  - Units

Equation-Based Object-Oriented Modelling of the Physics, with Modelica

- Programming: procedural code (function/algorithm)
- Equation-based (a-causal) modelling
- Behind the scenes: numerical approximations
- Object-Oriented modelling
- Libraries and the MSL
- Controller Modelling
- Extra time: Hiding IP: Composition of Functional Mockup Units (FMI)

PID Controller



Closed-loop system: better stability

P control by itself is unable to get rid of the steady-state error, which results in a permanent offset.

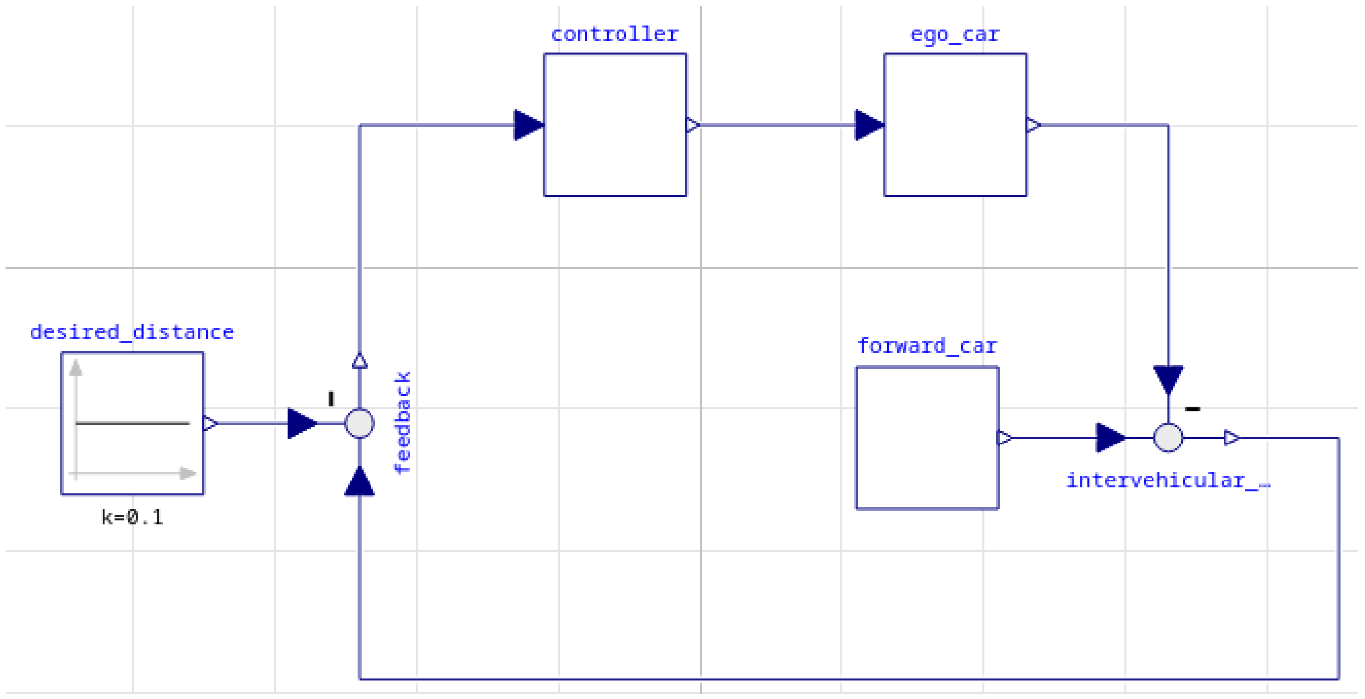
The steady-state error is eliminated by the integral component, which gradually accumulates the error and modifies the controller’s output. However, it may result in instability and oscillations from excessive integral activity.

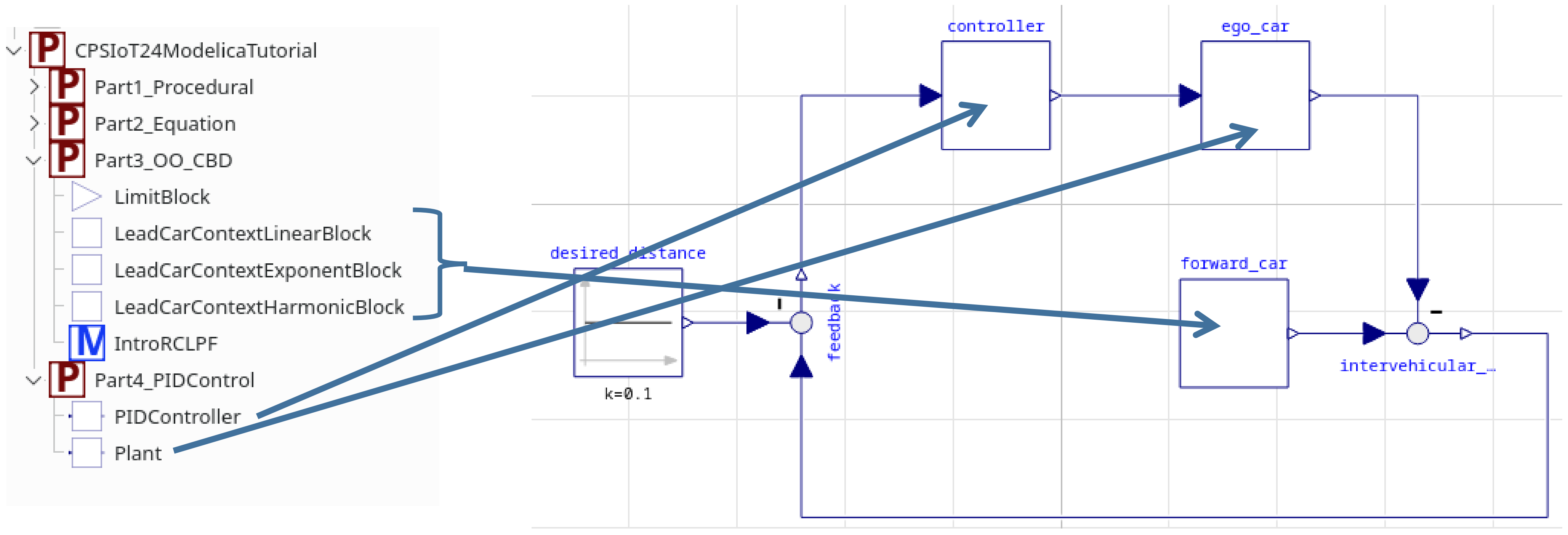
The derivative component forecasts the inaccuracy in the future. By increasing the derivative gain (Kd) by the error’s derivative over time, it produces a damping effect. By doing this, the response is smoothed down and oscillations and overshoot are lessened.

https://www.wattco.com/2024/05/pid-controller-explained/

rem. time

Given what you have learnt today, and considering that all blocks are provided. Can you now make the following PID control loop model of the robot to simulate its behavior?





What are the best values for Kp, Ki, Kd ??

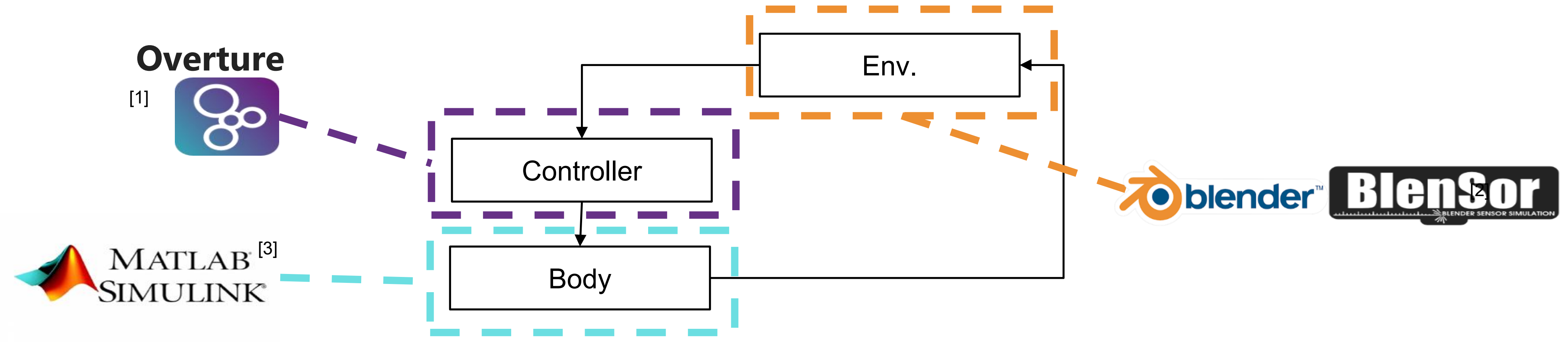
Remember these values, you will use them in the 2nd tutorial !

Equation-Based Object-Oriented Modelling of the Physics, with Modelica

- Programming: procedural code (function/algorithm)
- Equation-based (a-causal) modelling
- Behind the scenes: numerical approximations
- Object-Oriented modelling
- Libraries and the MSL
- Controller Modelling
- Extra time: Hiding IP: Composition of Functional Mockup Units (FMI)

problem: **full-system analysis**

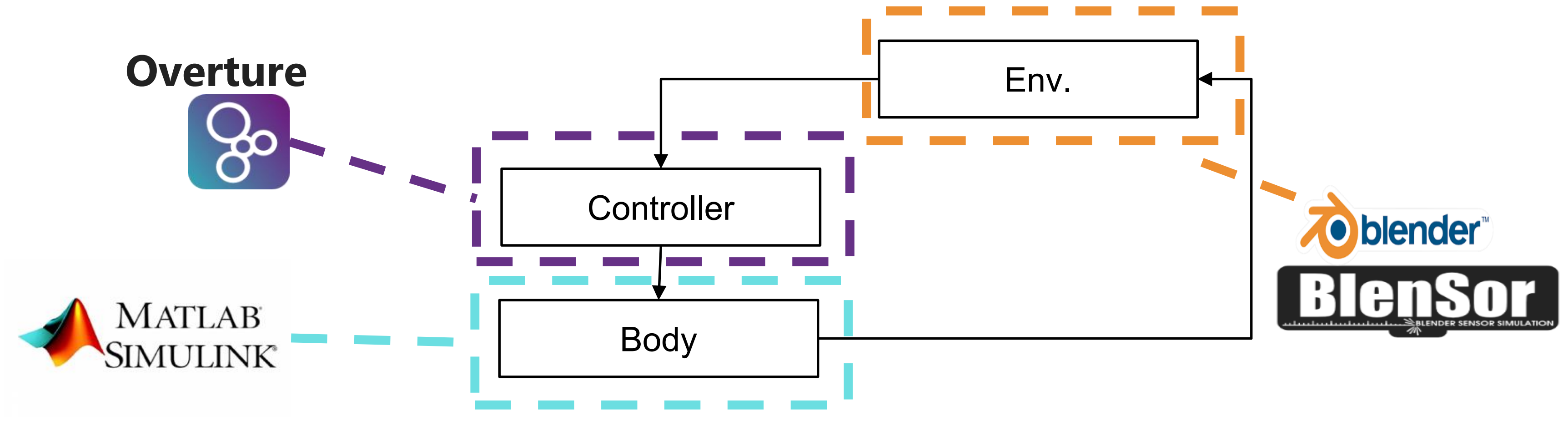
(also when IP protected)

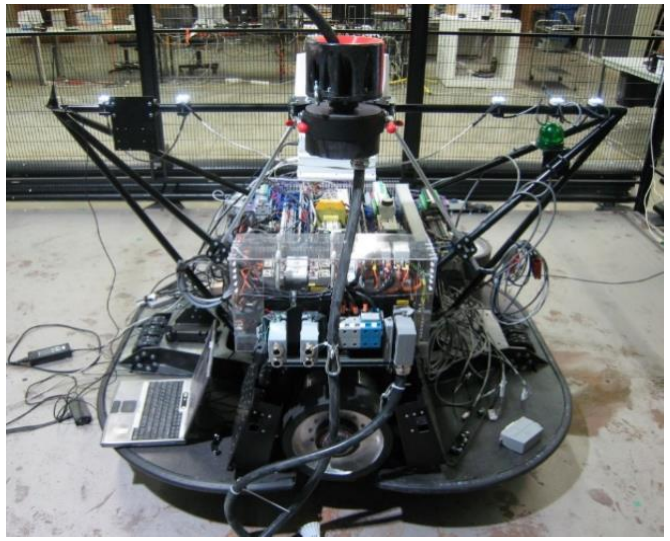


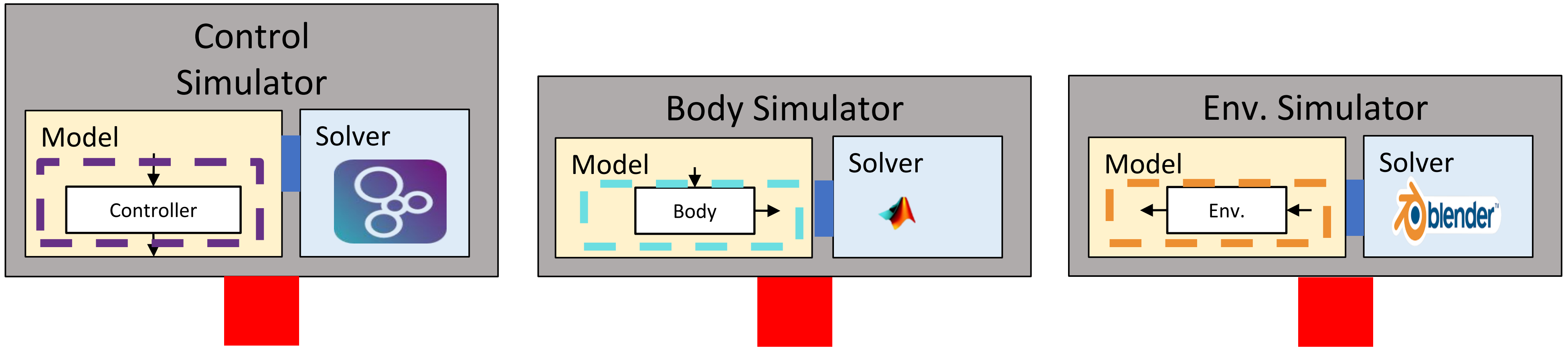
solution: combine sub-system **simulators**

aka **co-simulation**

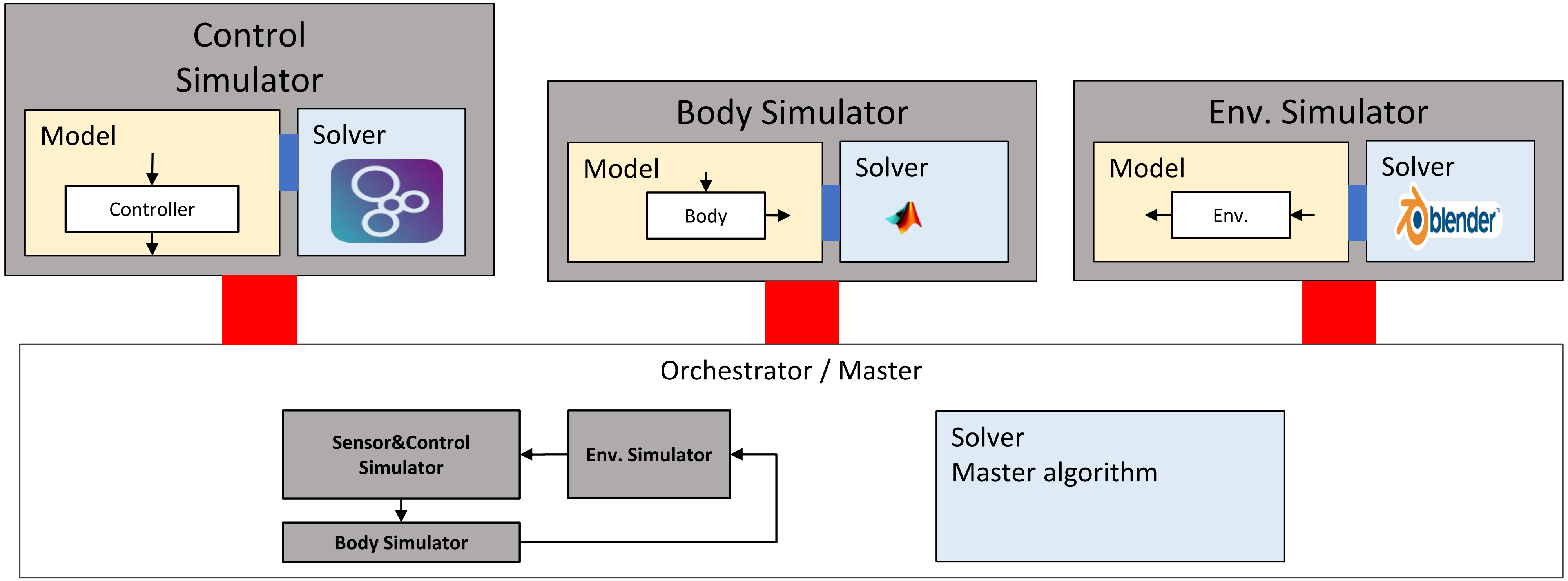
co-simulation: how? (when IP protected)

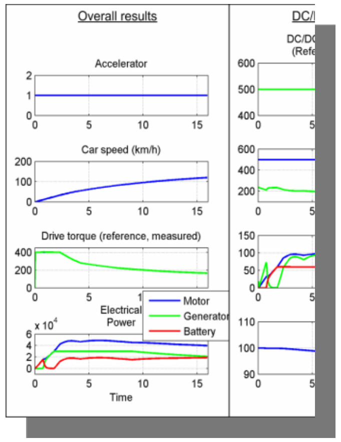






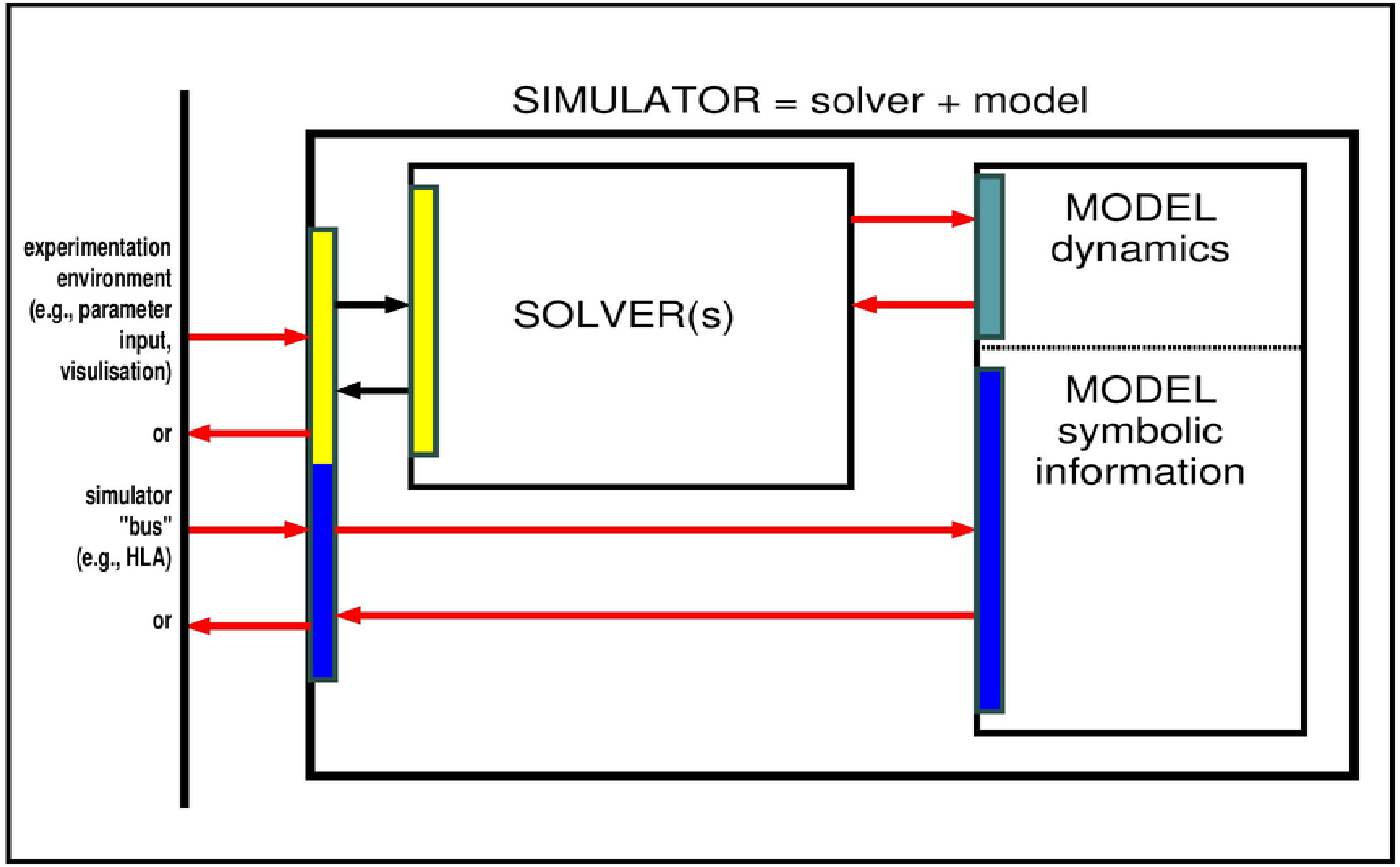
co-simulation: how? (when IP protected)





[Minimally, Constrained Stable Switched Systems and Application to Co-Simulation]
C Gomes, RM Jungers, B Legat, H Vangheluwe 2018
IEEE Conference on Decision and Control (CDC), 5676-5681

Model-Solver Interface
Simulator-Environment Interface

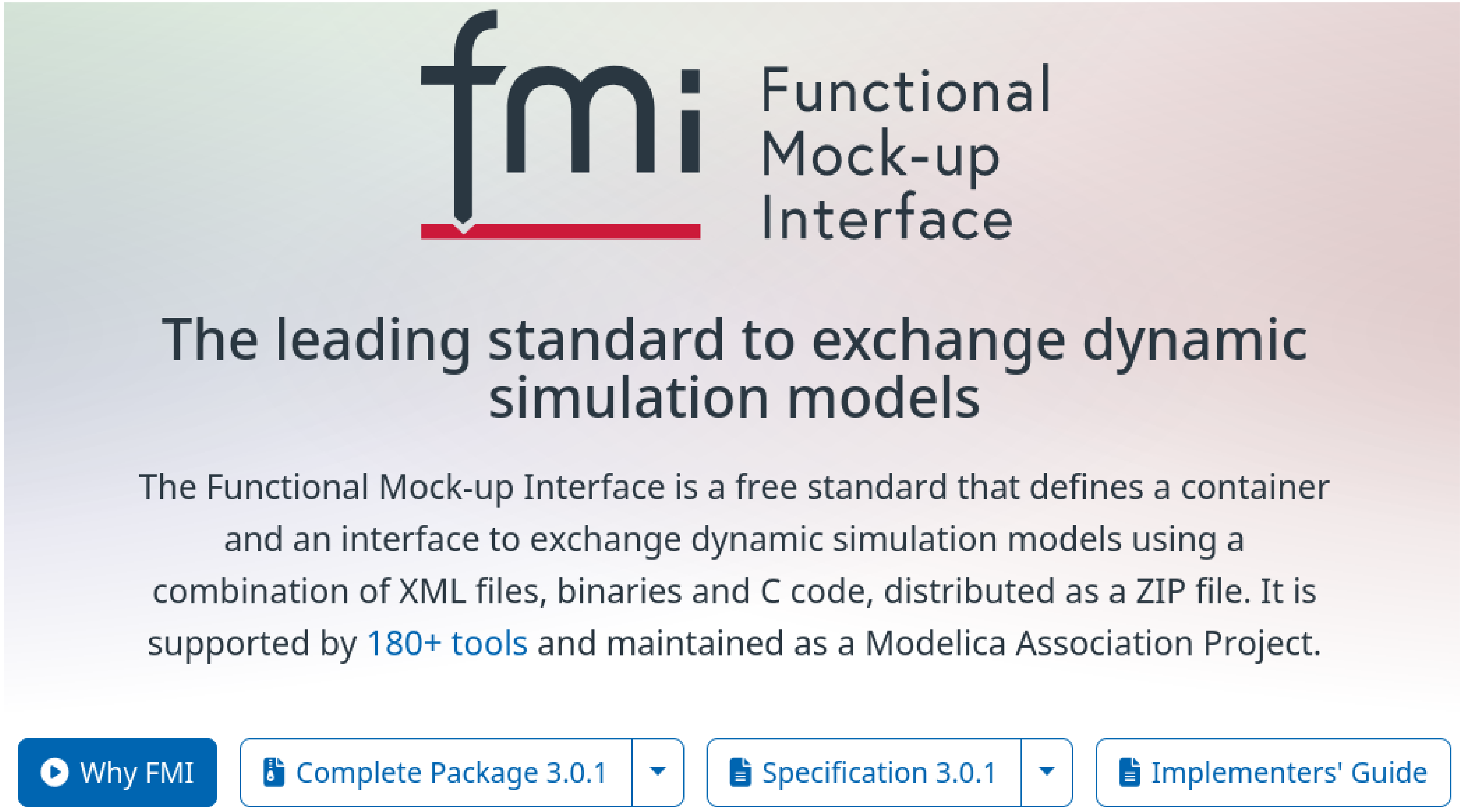

SIMULATOR = solver + model
experimentation environment (e.g., parameter input, visulisation)
or
simulator "bus" (e.g., HLA)
or
SOLVER(s)
MODEL dynamics
MODEL symbolic information

DSblock

MSL-EXEC

Martin Otter and Hilding Elmquist.
The DSblock interface for exchanging model components. Eurosim '95 Simulation Congress. pp. 505- 510. 1995.

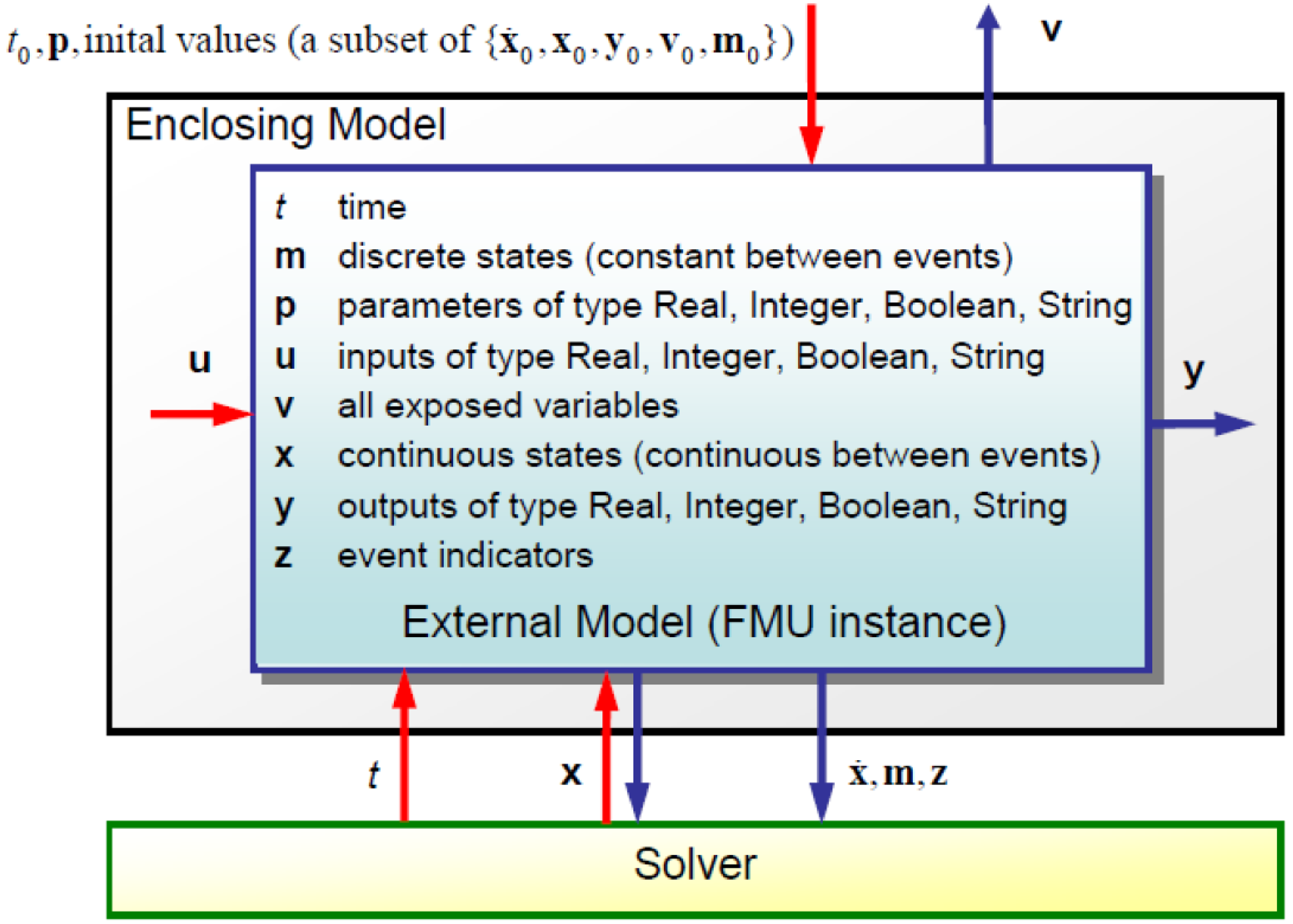
Henk Vanhooren, Jurgen Meirlaen, Youri Amerlinck, Filip Claeys, Hans Vangheluwe, and Peter A. Vanrolleghem.
WEST: Modelling biological wastewater treatment. Journal of Hydroinformatics , 5(1):27--50, 2003.



The leading standard to exchange dynamic simulation models

The Functional Mock-up Interface is a free standard that defines a container and an interface to exchange dynamic simulation models using a combination of XML files, binaries and C code, distributed as a ZIP file. It is supported by 180+ tools and maintained as a Modelica Association Project.

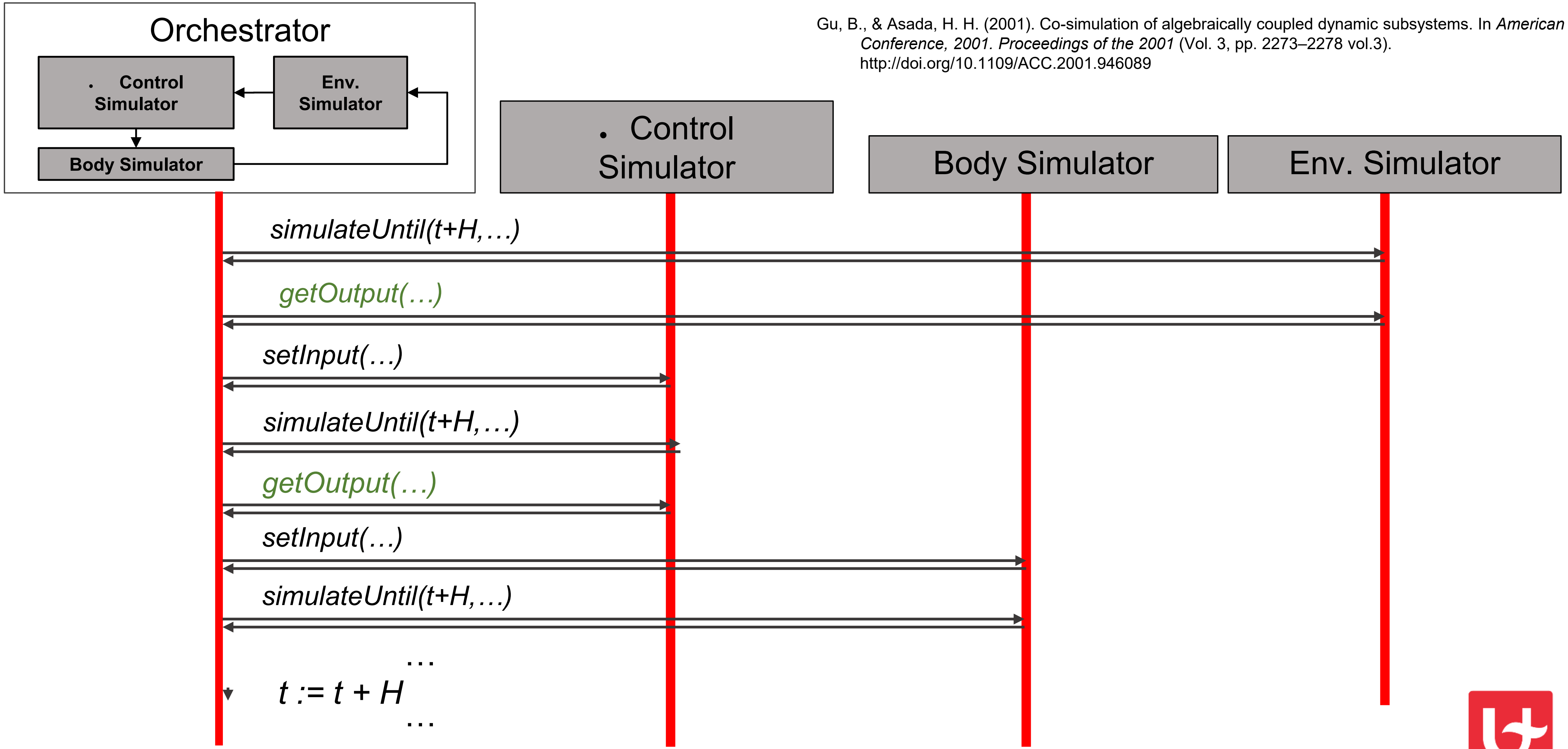
https://fmi-standard.org/


$t_0, \mathbf{p}$, inital values (a subset of $\{\dot{\mathbf{x}}_0, \mathbf{x}_0, \mathbf{y}_0, \mathbf{v}_0, \mathbf{m}_0\}$)
v
Enclosing Model
u
y
External Model (FMU instance)
t
x
$\dot{\mathbf{x}}, \mathbf{m}, \mathbf{z}$
Solver

t	time
m	discrete states (constant between events)
p	parameters of type Real, Integer, Boolean, String
u	inputs of type Real, Integer, Boolean, String
v	all exposed variables
x	continuous states (continuous between events)
y	outputs of type Real, Integer, Boolean, String
z	event indicators

Co-simulation: how?

Gu, B., & Asada, H. H. (2001). Co-simulation of algebraically coupled dynamic subsystems. In *American Control Conference, 2001. Proceedings of the 2001* (Vol. 3, pp. 2273–2278 vol.3). http://doi.org/10.1109/ACC.2001.946089



2 hands-on tutorials with foundations in Multi-Paradigm Modeling

Case Study: Adaptive Cruise Control System (ACCS)

1a: Modeling the ACCS using OpenModelica

Rakshit Mittal[1], Hans Vangheluwe[1]

1b: Verifying ACCS Safety Requirements using UPPAAL

Rizwan Parveen[2]

2a: Modeling and Analyzing the Architecture of the ACCS controller using AADL

Dominique Blouin[2], Anish Bhobe[3]

2b: Synthesizing Code for the ACCS controller using RAMSES

Dominique Blouin[2], Anish Bhobe[3]

[1]University of Antwerp – Flanders Make, Belgium

[2]Telecom Paris, France

[3]Institut Polytechnique de Paris, France

Understanding Model-driven Design with UPPAAL Model Checker

AGENDA

- Introduction to the basic concepts of modelling and model checking.
- Get to know basic features of the UPPAAL model checker.
- Illustration of UPPAAL tool through a few examples in the context of the formal verification

OUTLINE

1. The role of Model Checking in design validation

2. The UPPAAL Tool

    1. Introduction
    2. Building model and formalizing properties
    3. Verification: writing queries
    4. An example
    5. Installation instructions

3. References

OUTLINE

1. The role of Model Checking in design validation

2. The UPPAAL Tool

   1. Introduction
   2. Building model and formalizing properties
   3. Verification: writing queries
   4. An example
   5. Installation instructions

3. References

1. WHY DESIGN VALIDATION?

- Design Validation is important step to **ensure design correctness** at very early phase of SDLC

- **Traditional Techniques**:
  - **Simulation** (on an abstraction or a model of the system)
  - **Testing** (often conducted on the actual product once built)

- **Formal Methods** (aimed at exhaustive validation)

  - different formal approaches are used for different kind of requirements.

  - The complexity of these methods made them only accessible to specialists (mathematicians).

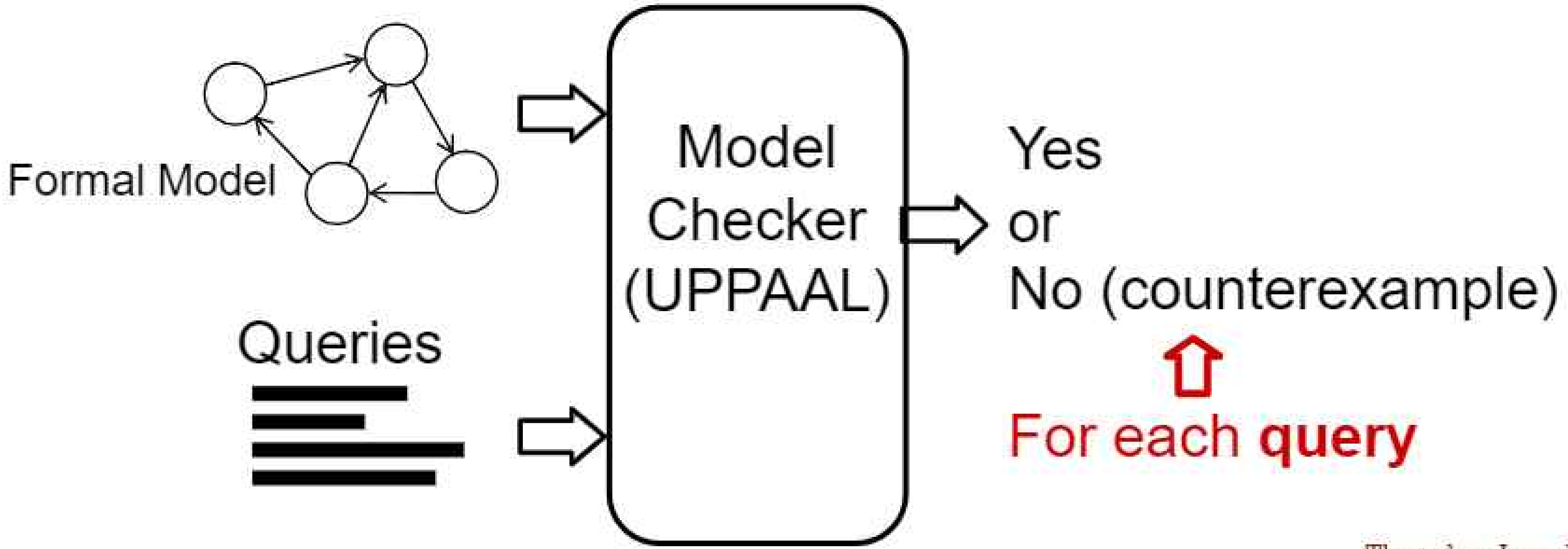
  - **Model Checking (MC)**
  - MC is the first technique that is truly accessible for "normal" engineers
  - Applicable to (finite-state concurrent systems ➔ automatic) sequential circuits, communication protocols, software… a wider spectrum of applications

PERFORM 3 STEPS FOR VERIFICATION

First, build a **model** for the system (abstract), in the form of a set of automata (called as Network of automata in UPPAAL)

Second, write the important **properties** to be verified using expressions, e.g. temporal logic (in case of UPPAAL, it is TCTL)

Third, use the model checker (a **tool like UPPAAL**) to generate the space of all possible states and to exhaustively check whether a property hold in each and everyone of the possible BEHAVIOURS of the model.



OUTLINE

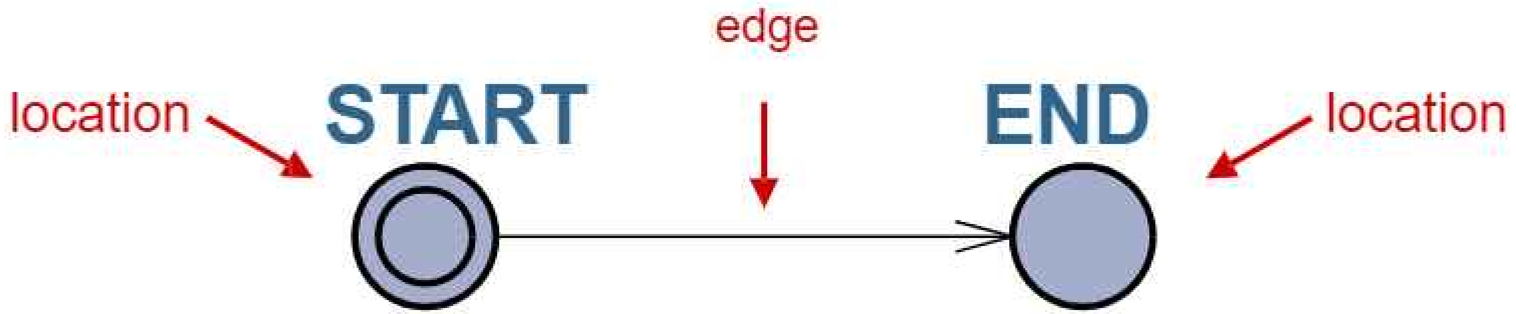
1. The role of Model Checking in design validation

2. The UPPAAL Tool

    1. Introduction
    2. Building model and formalizing properties
    3. Verification: writing queries
    4. An example
    5. Installation instructions

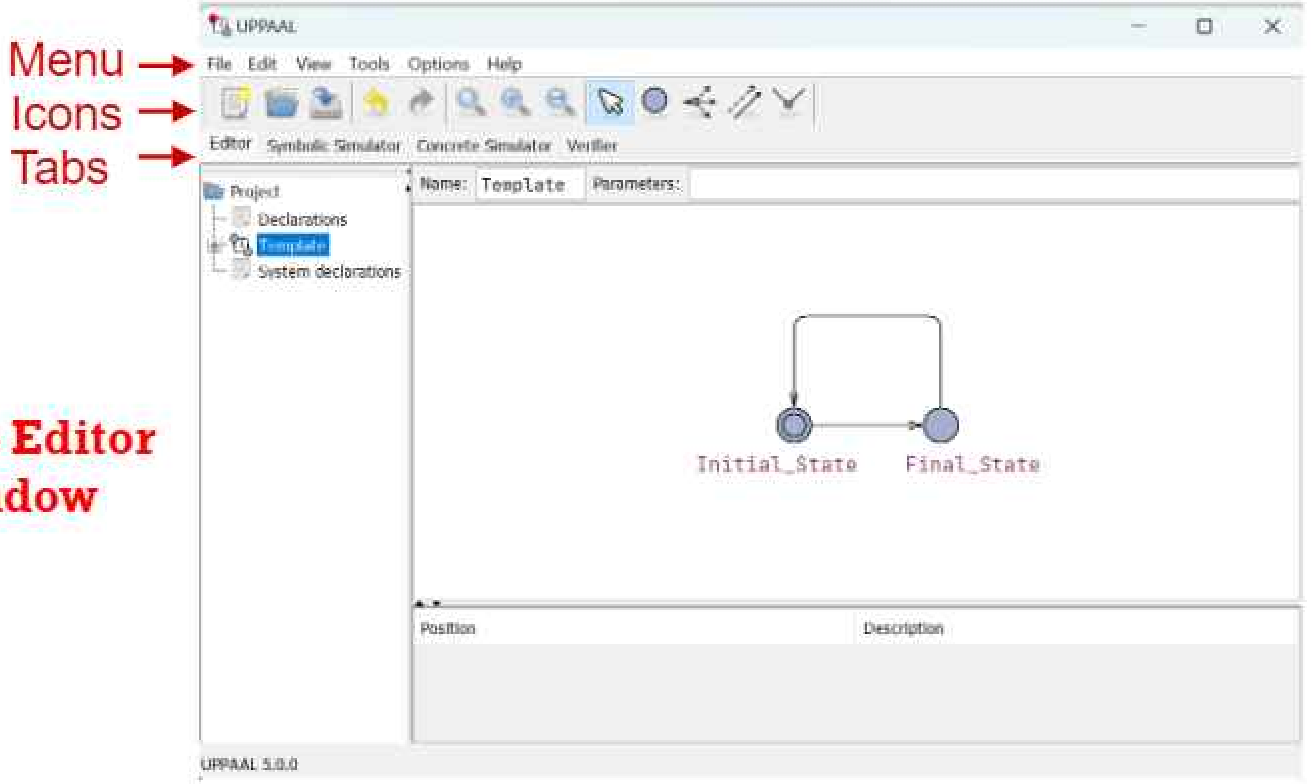
3. References

2. UPPAAL



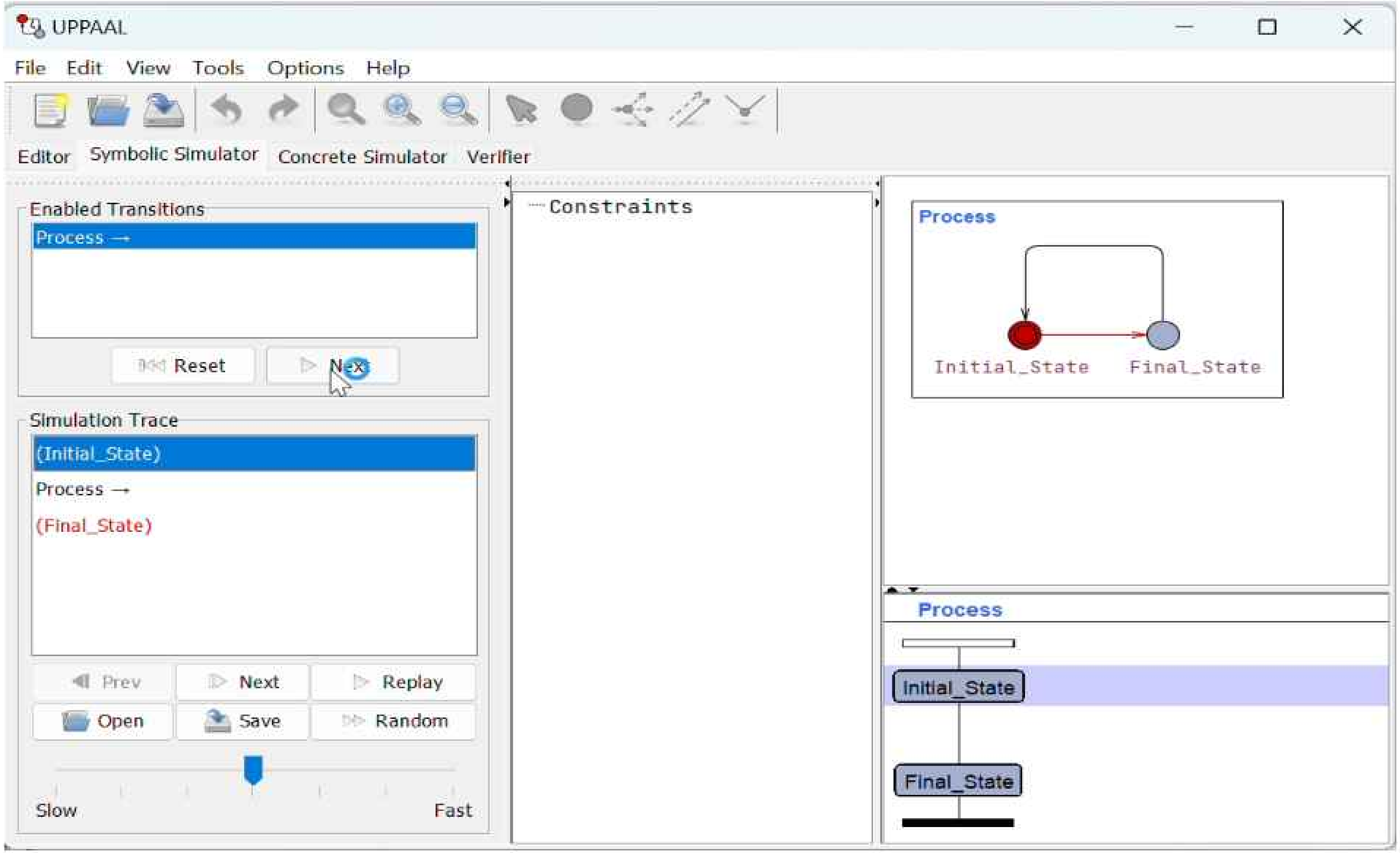
- Enable verification via automatic model- checking.
- It consists of three main parts:
  - a Graphical editor (run on the user's computer) and
  - a simulator
  - a verifier

All constitutes to a model-checker engine (by default executed on the same computer as the user interface, but can also run on a more powerful server)



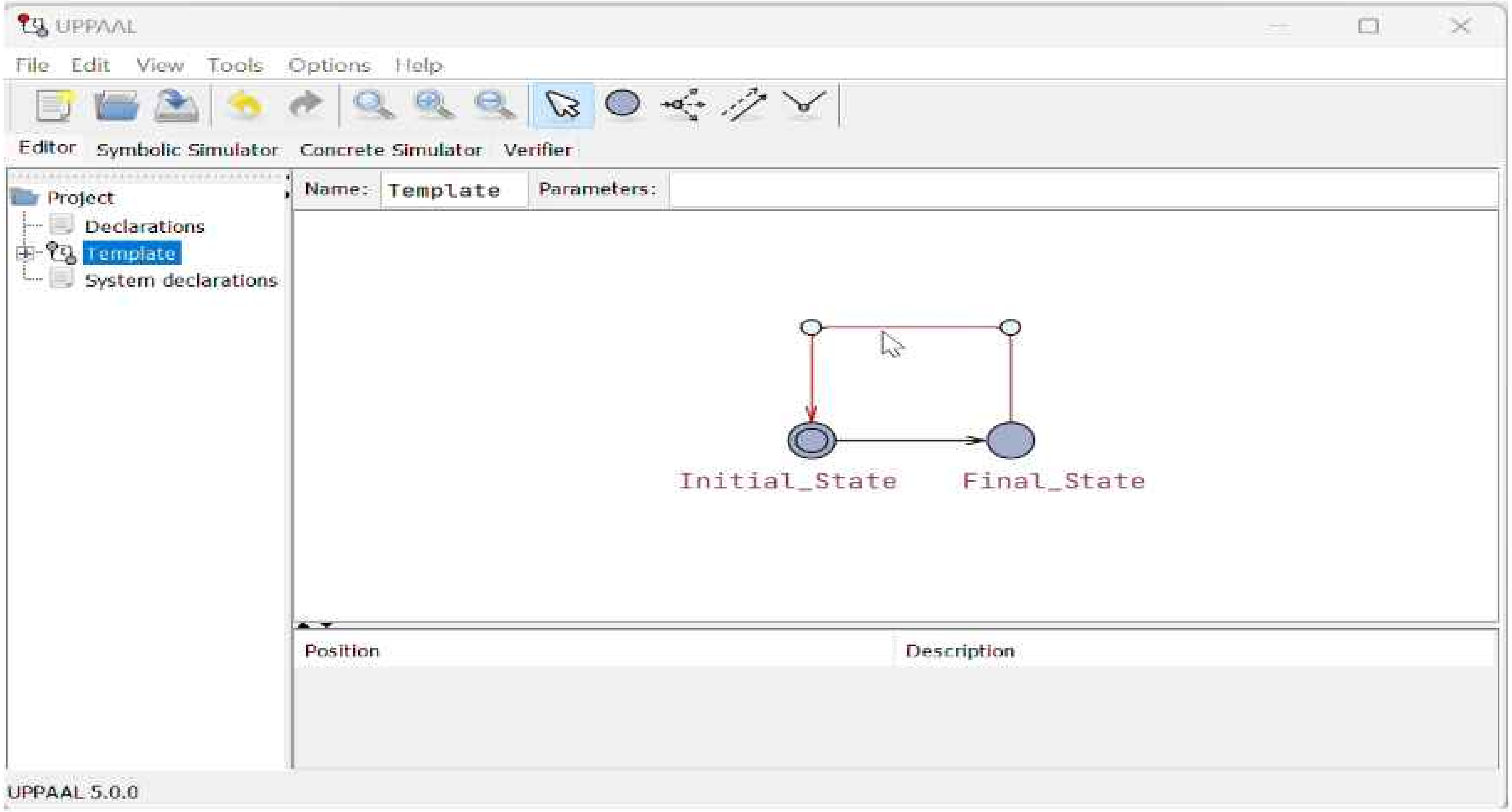
The Editor Window

THE SIMULATOR WINDOW

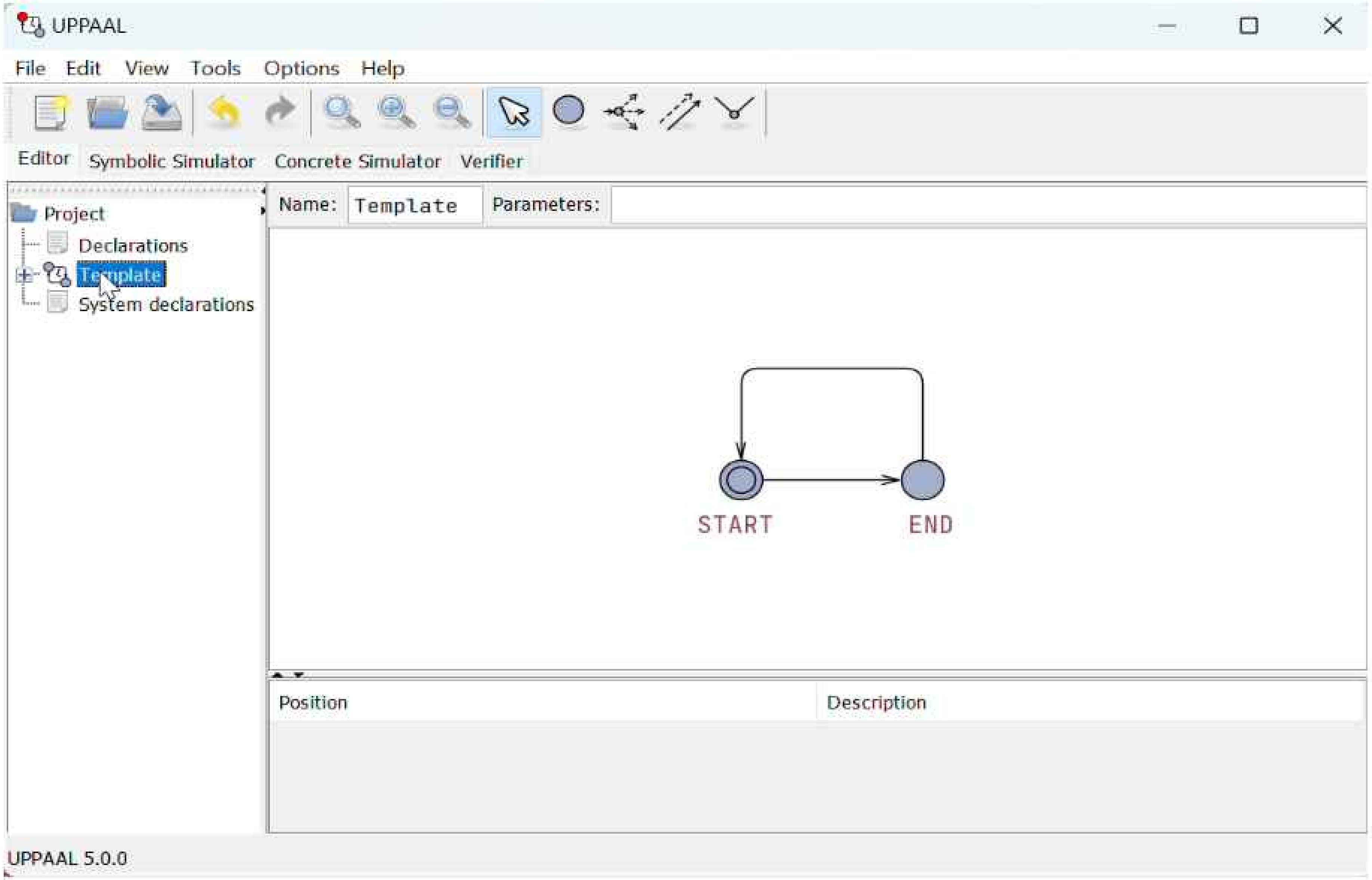


EDIT THE MODEL AND VERIFY

- An UPPAAL model is built as **a set of concurrent *processes***.
- Each process is graphically designed as a ***timed- automaton***.



THE VERIFIER WINDOW: INSERT QUERY


UPPAAL
File Edit View Tools Options Help
Editor Symbolic Simulator Concrete Simulator Verifier
Project
Declarations
Template
System declarations
Name: Template Parameters:
START
END
Position Description
UPPAAL 5.0.0

OUTLINE

1. The role of Model Checking in design validation

2. The UPPAAL Tool

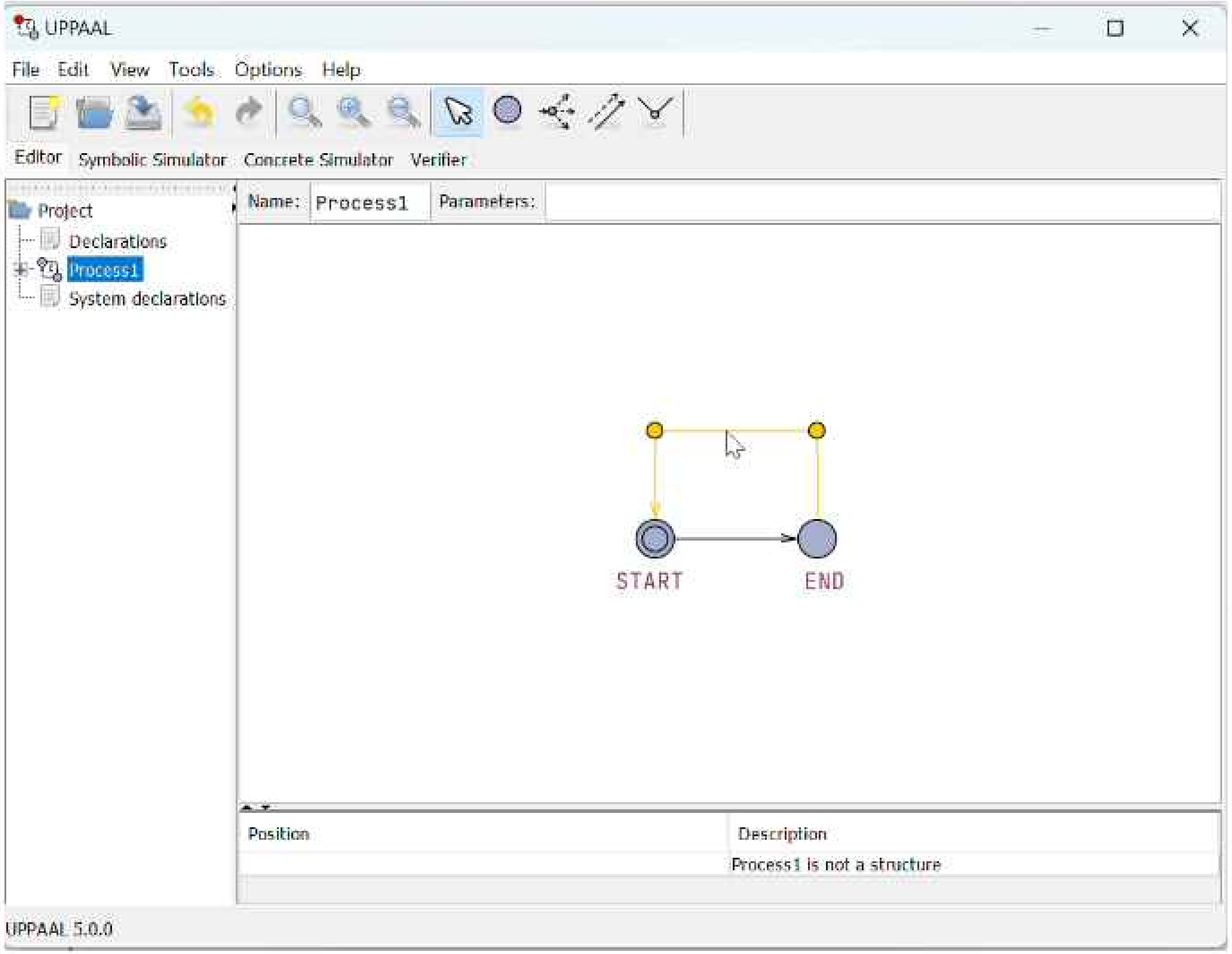
    1. Introduction
    2. Building model and formalizing properties
    3. Verification: writing queries
    4. An example
    5. Installation instructions

3. References

2. MODELLING WITH UPPAAL
Synchronisations: Guard and channels

- Edges are annotated with ***selections, guards***, ***synchronisations*** and ***updates***

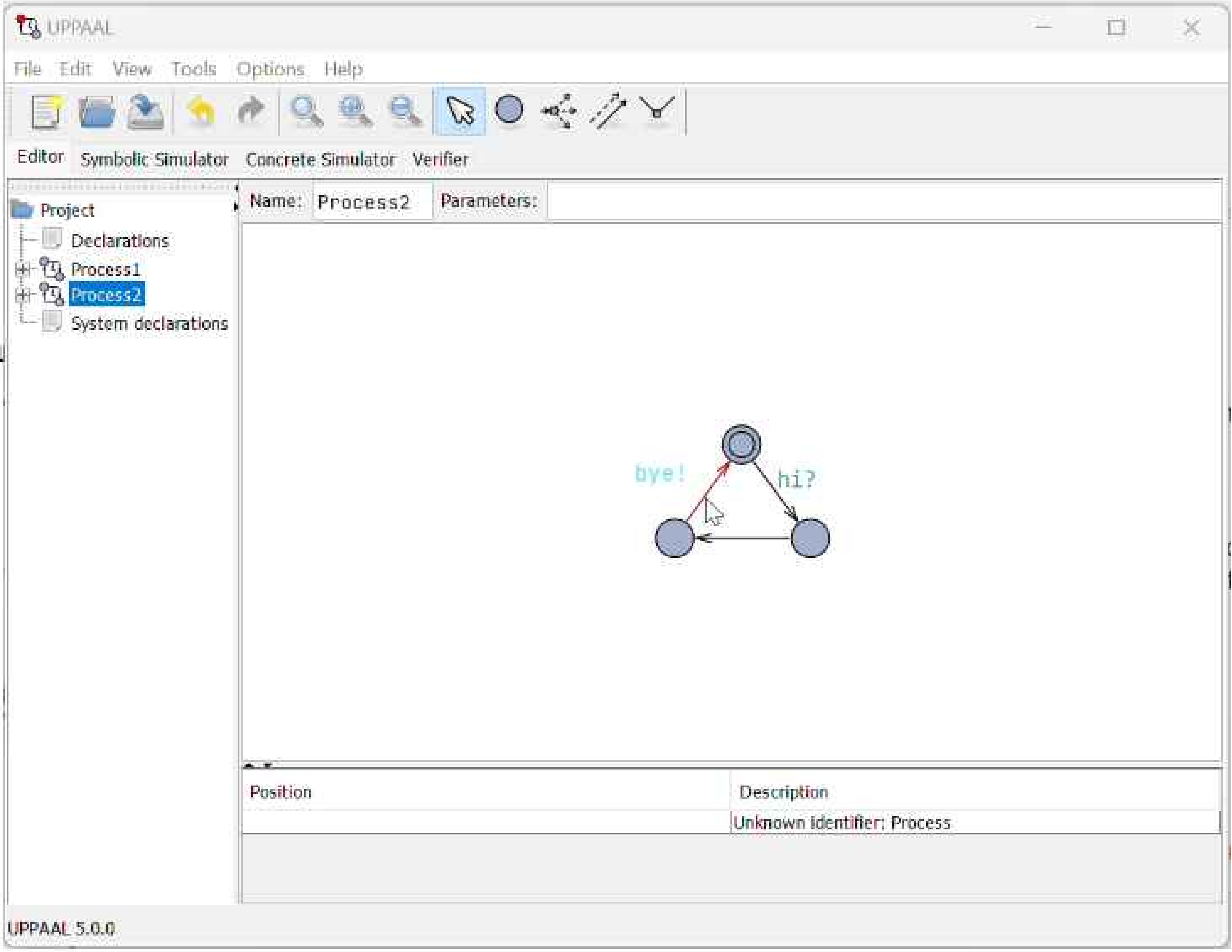
- Using **channels** two (or more) processes to take a transition at the same time.
- Declare the **channel** (*c*) under declaration using keyword ***chan***.
- One process will have an edged annotated with *c**!*** (*send*) and the other(s) process(es) another edge annotated with *c**?*** (*receive*)



Process1 is not a structure

SYNCHRONISATIONS : GUARD AND CHANNELS

- If at a specific instant there are several possible ways to have a pair *c!* and *c?*, one of them is non-deterministically chosen during model checking.



Position	Description
	Unknown identifier: Process

COUNTEREXAMPLE AND DIAGNOSTIC TRACE

This example will show:
A. how an error in model can be traced.

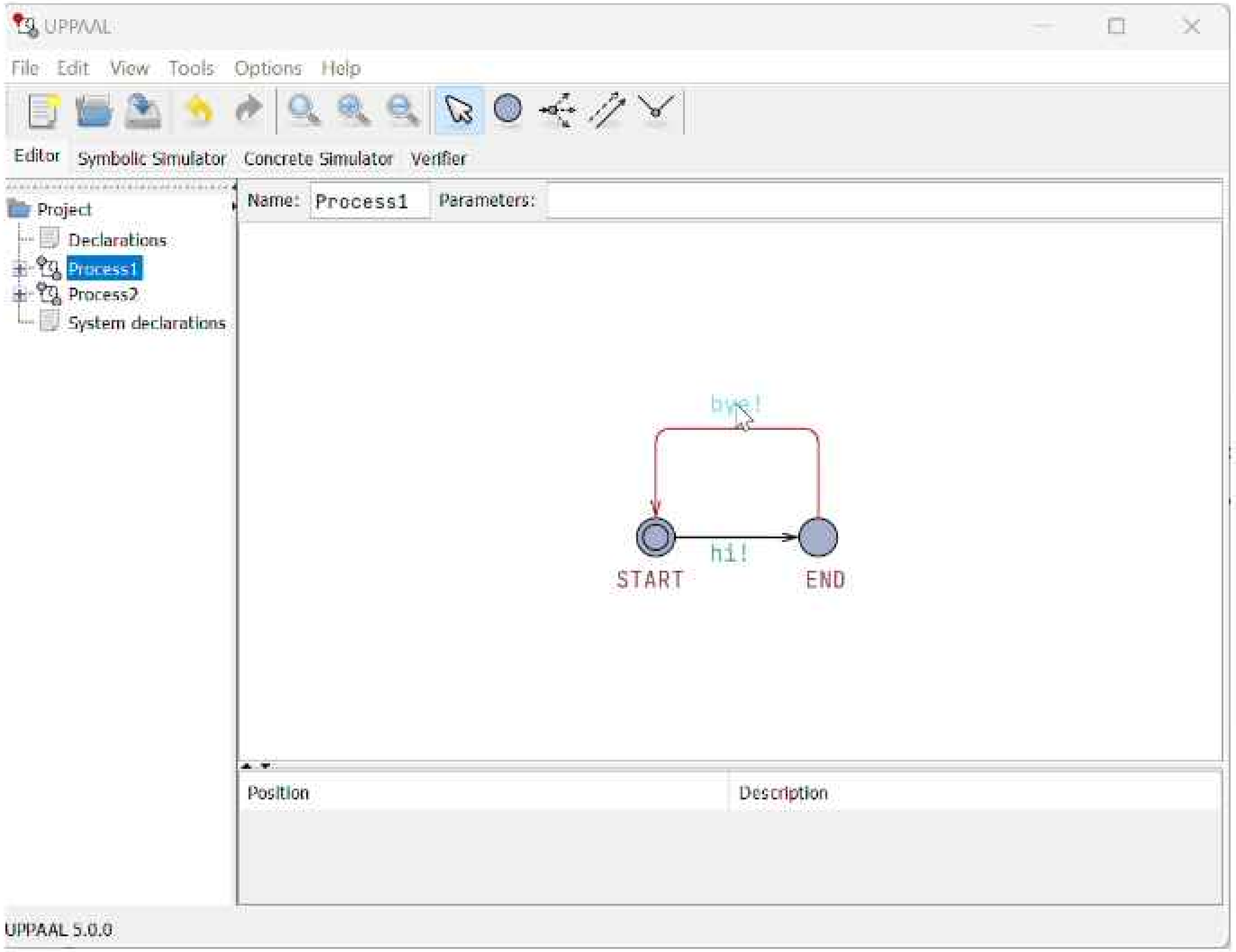
B. How to formalize query in TCTL.

- Verifying Properties:

1. to ensure that the model behaves as the system we wanted to model.
2. to detect some errors in the original design)

Formalize properties:

- Ex. In a network protocol, if a message is sent, it will be eventually received.

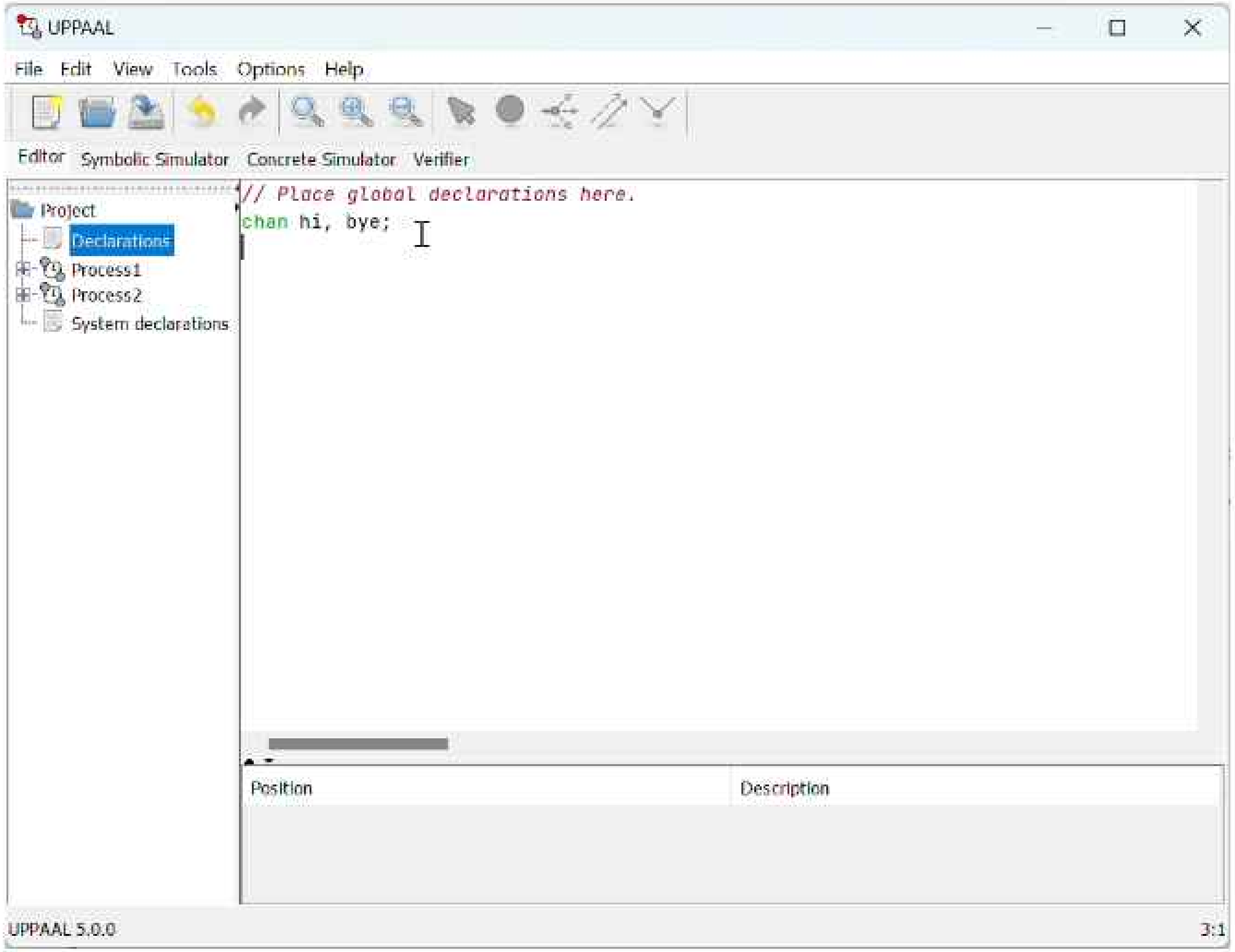


UPPAAL understands Timed Computational Tree Logic (TCTL). That means it is required to formalize those properties in TCTL (similar to LTL/CTL)

UPDATE AND GUARD

- **A guard is** an expression (a condition/action on the transition).
- It uses the variables and clocks of the model in order to indicate when the transition is enabled or not.
  - Note that several edges may be enabled at an specific time but only one of them will be fired ➜ leading to different potential **interleavings**

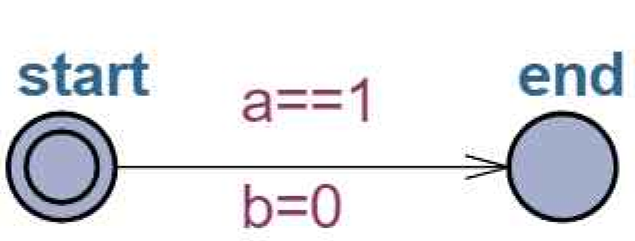
An update is an expression that is evaluated as soon as the corresponding edge is fired. This evaluation changes the state of the system.

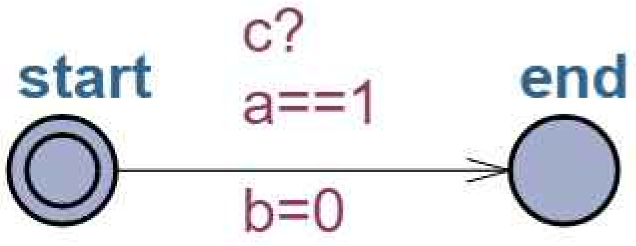


```
// Place global declarations here.
chan hi, bye;
```

EDGES

- **Three different kinds of synchronizations**:
  - **Regular channel** (leading to Binary Synchronization)
  - **Urgent channel:** time cannot lapse
  - **Broadcast channel:** all these transitions are enabled at receiving ends.





- The update expression on an edge synchronizing on *c!* is executed **before** the update expression on an edge synchronizing on *c?*

STATES (AKA LOCATIONS)

- **States can be of three different types** (that can be assigned by double-clicking on the location):
  - **Initial**
  - **Urgent** (time is not allowed to pass when a process is in an urgent location)
  - **Committed** (When a model has one or more active committed locations, no transitions other than those leaving said locations can be enabled)
  - **Normal** (all the rest)

A RECOMMENDATION ON MODELING

- **The state space grows very quickly** with the model complexity (state space explosion). It is necessary to:
  - It is better to model at suitable level of abstraction of a system.
  - Identify important properties to model and properties that are essential to be verified.

- More specifically:
  - The use of committed locations can reduce significantly the state space, but it can possibly take away relevant states.
  - The number of clocks and variables

This is rather an "art" (model checking may not be so "perfect" but it helps a designer to think)

OUTLINE

1. The role of Model Checking in design validation
2. The UPPAAL Tool
   1. Introduction
   2. Building model and formalizing properties
   3. Verification: writing queries
   4. An example
   5. Installation instructions
3. References

VERIFICATION AND TYPES OF QUERIES IN UPPAAL

The UPPAAL query language (TCTL) can be classified as:

[1] ***Reachability properties****. A specific condition holds in some state*. `Expressed as : E<> p` "Exists eventually p"

[2]. ***Safety properties****. A specific condition holds in all the states of an execution path.*

`E[] p` *"Exists globally* `p`*" (p holds for all the states of the path)*

`A[] p` "Always globally `p`" (For each (all) execution path p holds for all the states of the path)

[3]. ***Liveness properties****. A specific condition is guaranteed to hold* ***eventually*** *(= at some moment)*

`A<> p` "Always eventually p" (p holds for at least one state of the path)

q-->p "q always leads to p"

[4]. ***Deadlock properties****. If a deadlock is possible or not in the model*

`A[] not deadlock`

OUTLINE

1. The role of Model Checking in design validation

2. The UPPAAL Tool

    1. Introduction
    2. Building model and formalizing properties
    3. Verification: writing queries
    4. An example
    5. Installation instructions

3. References

MOVEMENT OF A CAR

1. Avoiding Obstacle
2. Maintaining safe distance from the vehicle in front

To avoid obstacle, there are two actions:

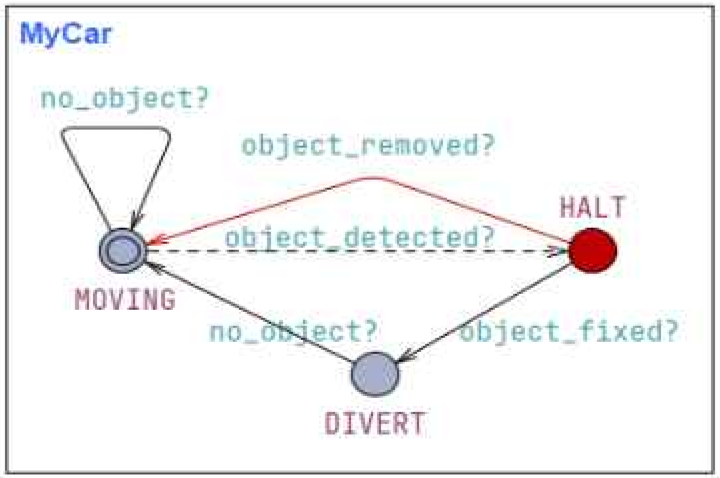
1. Slow down the speed of the car

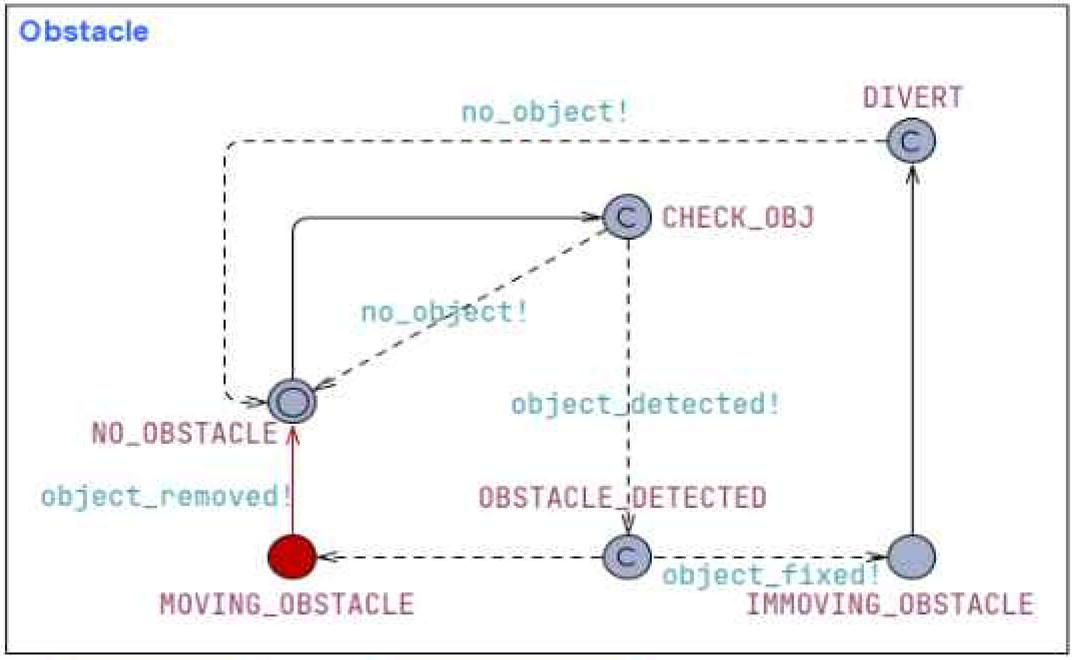
2.1 If it is movable obstacle, wait till the obstacle is removed from the path and resume moving.

2.2. If it is non-movable obstacle, wait and divert the path.

In a advanced model, we can add path planning/shortest part, etc. algorithm from the state of "divert".

1. AVOIDING OBSTACLE



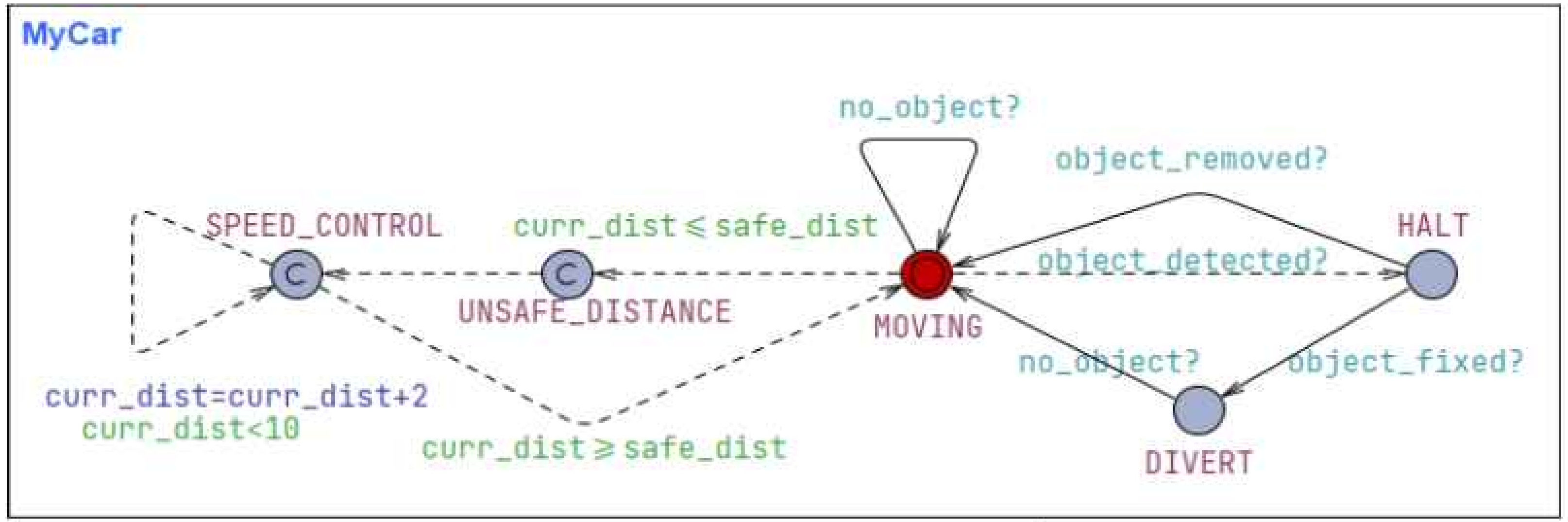


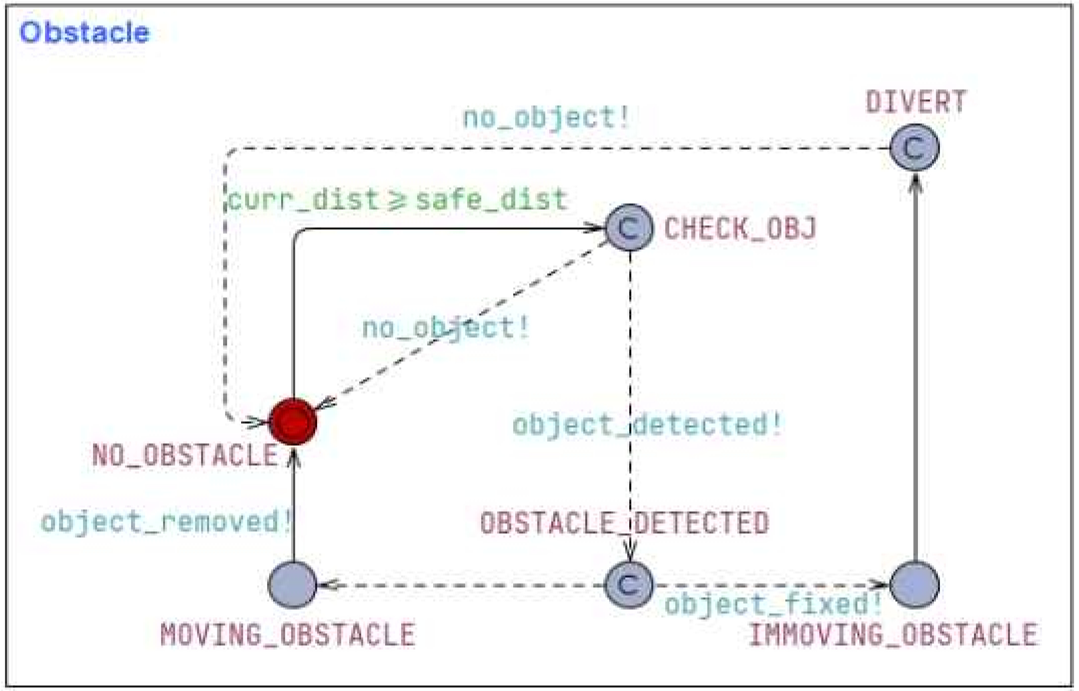
- The model shows two automata: MyCar and Obstacle
- Assume my car is in moving state. It keeps moving until it detects an obstacle.
- In the event of a obstacle detected, my car has two options:
  - A. To wait for obstacle to move away from the path and then continue moving on the path
  - OR B. My car chooses a different path and resume moving.

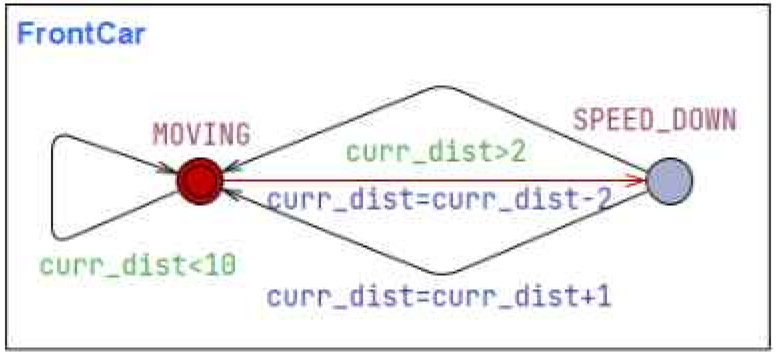
2. MAINTAINING SAFE DISTANCE FROM THE VEHICLE IN FRONT

- We add one automaton in the existing model to represent the operation of a front car.
- Let's assume if this front car slows down its speed, maybe during a heavy traffic, that means the distance between my car and front car will be reduced and not in a safe range.
- There is a minimum safe-distance which my car has to maintain from the front car. Therefore, whenever the front car reduces the speed, my car checks if it is moving on a safe distance or not.
- If not, my car control its speed (reduce) and go to safe moving only when safe distance is recovered (that represented by FrontCar's normal moving state).

2. MAINTAINING SAFE DISTANCE FROM THE VEHICLE IN FRONT







VERIFICATION

- Check for deadlock
- Check that MyCar should not be in MOVING state when obstacle detected.
- Check MyCar always maintain safe distance from the FrontCar

OUTLINE

1. The role of Model Checking in design validation

2. The UPPAAL Tool

    1. Introduction
    2. Building model and formalizing properties
    3. Verification: writing queries
    4. An example
    5. Installation instructions

3. References

LEARNING OBJECTIVE

- How to build model with UPPAAL?
- Identifying important properties and formalizing them.
- Verify important properties of the model.

Task to be performed:

- Follow the UPPAAL installation instruction given on next slide.
- Download the pre-build model of the car.
- Improve this model by implementing task #2: maintaining safe distance
- Write Safety properties and verify them

INSTALLATION INSTRUCTIONS

- Make sure you have the Java version installed as per latest UPPAAL requirement.
  - E.g.: www.java.com/es/download/manual.jsp
- Go to the UPPAAL page: www.uppaal.org
- Click on the download tag and then on the link Uppaal 5.0 (current official release)

LINK: https://uppaal.org/downloads/#uppaal5.0

- Fill the (academic) license agreement form. Click on "Register & Download". You may need to provide your university email id to get this license.
- Unzip files
- To run UPPAAL double-click the file uppaal.jar

REFERENCES

Some of the following references are used for creating this presentation and some useful for further reading

- Slide Credit: Julián Proenza, Systems, Robotics and Vision Group. UIB. SPAIN

- UPPAAL (available at *www.uppaal.org*)
  - *A Tutorial on Uppaal,* 17 Nov 2004 by G. Behrmann, A. David, and K. G. Larsen.
  - UPPAAL Online Help

- Model Checking:

- Behrmann, G., David, A., Larsen, K.G. (2004). A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds) Formal Methods for the Design of Real-Time Systems. $SFM-RT$ 2004. Lecture Notes in Computer Science, vol 3185. Springer, Berlin, Heidelberg.

- Bouyer, Patricia (2009). "Model-checking Timed Temporal Logics". In: Electronic Notes in Theoretical Computer Science 231. Proceedings of the 5th Workshop on Methods for Modalities(M4M5 2007), pp. 323–341. ISSN: 1571-0661.