Automated Verification of Cyber-Physical Systems

A.A. 2024/2025

Corso di Laurea Magistrale in Informatica

Simulation of Systems

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Simulation vs Model Checking

- In "standard" Model Checking, we are given
  - a non-deterministic Kripke Structure (KS)
  - an LTL or CTL property to be verified
- The output is either PASS or FAIL
  - if PASS, then *all* evolutions (paths) of the given model fulfill the given property
  - if FAIL, we also have a counterexample
- In probabilistic model checking, we consider probabilities of sets of evolutions
- *Simulation* of a system only considers *one* path

Murphi Simulation

```
void Make_a_run(NFSS 𝒩, invariant φ)
{
 let 𝒩 = ⟨S, I, Post⟩;
 s_curr = pick_a_state(I);
 if (!φ(s_curr))
  return with error message;
 while (1) { /* loop forever */
  if (Post(s_curr) = ∅)
   return with deadlock message;
  s_next = pick_a_state(Post(s_curr));
  if (!φ(s_next))
   return with error message;
  s_curr = s_next;
 }
}
```

SPIN Simulation

```
void Make_a_run(NFSS 𝒩)
{
 let 𝒩 = ⟨S, {s0}, Post⟩;
 s_curr = s0;
 if (some assertion fail in s_curr))
  return with error message;
 while (1) { /* loop forever */
  if (Post(s_curr) = ∅)
   return with deadlock message;
  s_next = pick_a_state(Post(s_curr));
  if (some assertion fail in s_curr))
   return with error message;
  s_curr = s_next;
 }
}
```

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Repeating a Simulation

- Simulations may be *deterministic* or *probabilistic*
  - both Murphi and SPIN simulations are probabilistic
  - at each step, a transition is chosen among the n possible ones with probability $\frac{1}{n}$
  - of course, n may be different at each step
- Running multiple probabilistic simulations typically implies obtaining different paths
  - the longest the path, the more likely this is to happen

Repeating a Simulation

- For deterministic simulations, all runs are the same
  - multiple simulations all result in the same path
- Deterministic simulation are however important when *inputs* from the environment are present
  - this is actually true for many systems
  - some inputs must be given an system startup, others must be given during the system evolution
- Running multiple simulation result in different paths if we vary the inputs to be received
  - this is actually true for many systems
- We of course may have inputs from the environment also in probabilistic simulations

Simulation

- Similar to testing
- If an error is found, the system is bugged
  - or the model is not faithful
  - actually, simulation in standard model checking is also used to understand if the model itself contains errors
- If an error is not found, we cannot conclude anything
- The error state may lurk somewhere, out of reach for the random choice in `pick_a_state`

Simulation vs Modeling

- However, for complex CPSs simulation is needed
- In fact, accurately modeling a complex CPS in a classical model checker is often too difficult or inconvenient
  - plant must be modeled by *real* variables: inherently infinite state systems
  - can be approximated, but accuracy may be low
- Simulators are often already available for testing, why can't we rely on them?
  - not "real" model checking, but something close to it
  - far better than "simple" testing
- May be either build from scratch, or implemented with dedicated tools
  - C/Java/Python dedicated programs vs. Simulink/Modelica

Simulation-based Model Checking

- Too many states, we cannot store them in an hash table
  - transition relation defined by a complex simulator, translation in OBDDs cannot be done
- Two main workhorses:

  System Level Formal Verification chooses system inputs so as to cover as much as possible
  - mainly for safety, but also some sort of LTL may be used

  Statistical Model Checking uses powerful statistical methods to perform model checking
  - something like Monte-Carlo sampling
  - i.e., we run the simulation several times, and we try to derive some guaranteed answer

Simulation

- A simulation is an experiment on a model
    - we focus on simulations performed by a computer
- Simulation is very easy to implement in the case of classical model checkers
    - no problems with RAM or execution time
- This stems from the fact that classical model checking deals with finite state systems
    - one step at a time, time passing typically not important
    - state space is finite and described by discrete-typed variables
    - computing a transition from a state to another is straightforward

Simulation

- What if we need to simulate a cyber-physical system?
  - e.g., simulate the Apollo mission
  - many subsystem interacting with each other via continuous signals
  - some subsystems are described by ODEs (Ordinary Differential Equations)
- In some cases, system developers also builds a simulator from scratch, e.g., in the C language
  - directly experimenting on the physical object may be dangerous, expensive, or simply impossible (e.g., it still does not exist)
- Many tools are available to easily describe complex models to be simulated
  - e.g., able to approximate solutions for ODEs
- Here we will deal with the open-source Modelica
  - we will also have a look to Simulink

Some Background: ODEs

- With some simplification, an ODE is an equation

$$F(x, y, y', y'', \ldots, y^{(n)}) = 0$$

- The unknown y is a function $y = f(x)$, and $y^{(n)}$ is the the n-th derivative of y w.r.t. x
- In our context, the independent variable is time, denoted by t
  - in simulations, we are interested in the system evolution over time...
- Thus, we have functions $x = f(t)$ and n-th derivatives expressed in Newton's notation $\overset{n}{\dot{x}}$
- Finally, our ODE is an equation

$$F(t, x, \dot{x}, \ldots, \overset{n}{\dot{x}}) = 0$$

Some Background: ODEs

- We will consider *explicit* ODEs $\dot{x} = F(t, x)$
  - x usually is in some n-dimensional space, e.g., $x \in \mathbb{R}^n$
  - thus, this is a system of equations
  - note that, with explicit ODEs, derivatives higher than 1 are not needed
  - simply put $x_1 = x, x_2 = \dot{x}_1 \ldots$

Example ODEs

- $\dot{x} = t + x$
- $(\dot{x}_1, \dot{x}_2) = (t + x_2 e^{x_1}, x_1 \log t)$
- Model of an infectious desease (HIV):
  $(\dot{x}_1, \dot{x}_2) = (\lambda - d x_1 - \eta\beta x_1 x_2, -x_2(a + I) + \eta\beta x_1 x_2)$
  - x_1, x_2 are uninfected and infected cells, I is an action by the immune system
  - $a, d, \lambda, \beta, \eta$ are system parameters
  - this latter example is time-invariant (see next slide)

ODEs: Euler Approximation

- Given the *time-invariant* ODE $\dot{x} = F(x)$, we may use the Euler approximation
  - for small τ, $\dot{x} \approx \frac{x(t+\tau)-x(t)}{\tau}$
  - if we sample time with τ, i.e., we only consider $t \in \{0, \tau, 2\tau, \ldots, k\tau, \ldots\}$...
  - ... we have that $F(x(k\tau)) = \frac{x(k\tau+\tau)-x(k\tau)}{\tau}$
  - thus by setting $x_k = x(k\tau)$, we have a discrete-time difference equation $x_{k+1} = x_k + \tau F(x_k)$
- This only works for small τ and small k
  - it can be proved that $||x_k - x(k\tau)|| \leq \tau\psi(k)$, where ψ is *not* bounded
  - at least, in the general case

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

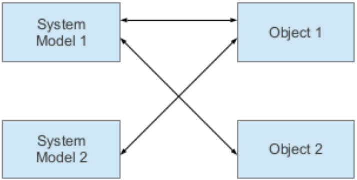
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Some Background: Systems

- A *system* is a mathematical concept used to study properties of physical objects
  - sometimes also called abstract system, or system model
- It is typically used to study evolutions as a function of *time*
- Virtually infinite examples:
  - population of rabbits
  - spread of diseases
  - physical objects: a fridge, an oven, a building, a car, ...
  - part of physical objects: a resistor, a brick, a wheel, ...
  - controllers for physical objects: ABS, autonomous driving, ...
- First distinction is among objects (what we want to model) and system (the mathematical model)
  - a system is defined through functions, sets, etc

Some Background: Systems

- Given an object, one may devise different systems
  - as we may have different programs to solve the same problem
  - not only because of different people doing it: different properties on the same object may be investigated
- Given a system, it may be applied to different objects
  - spreading of different diseases may have a common model
  - wheel of a car and of a motorcycle



Some Background: Systems

- We start defining systems by looking at their *inputs and outputs*
  - keeping in mind that it is all as a function of time
- Deterministic systems: given an input sequence from some "start", the output is the same
  - probabilistic systems also exist, we do not consider them here
- Black-box system: at first, we perform *experiments* on the system
  - we provide sequences of inputs and observe the sequence of outputs

Some Background: Systems

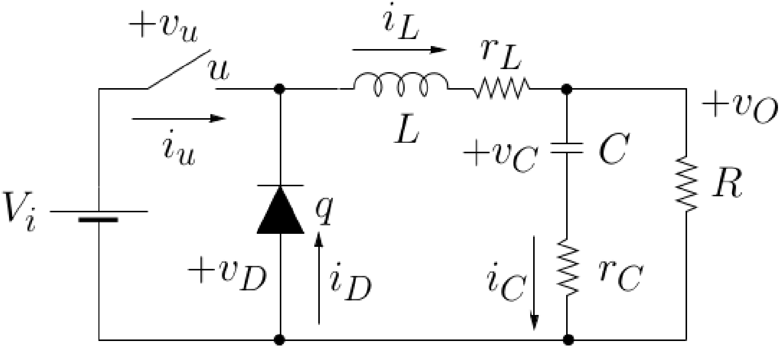
- We begin experiments at some time $t_0 \in T$, with $T \subseteq \mathbb{R}$
  - for some systems, $T \subseteq \mathbb{N}$
- We consider all input functions $u : T \to U$ for our object
  - U is some set on which inputs may vary
  - it may be multidimensional, e.g. $U = \mathbb{N} \times \mathbb{Z} \times \mathbb{R}^2$
  - of course, such input functions are uncountably many, this is a conceptual experiment
- For each u, we have an output function $y : T \to Y$ coming out of the object
  - U and Y may be different
  - again, Y may be multidimensional
- We define the system $\mathcal{S} = \{(u, y) \mid u$ is an input function and y the corresponding output function$\}$
  - thus, $\mathcal{S} \subset \mathcal{U} \times \mathcal{Y}$

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Some Background: Systems

- Example: determine the number of students which graduate in same bachelor course
  - assumption: student enrolls once every year, thus $T \subset \mathbb{N}$
  - $U \subset \mathbb{N}$: number of students enrolling "from outside"
  - $Y \subset \mathbb{N}$: number of graduated students
  - $\mathcal{U} = \{f \mid f : T \to U\}$, analogous for $\mathcal{Y}$
  - example of input-output:
    - $u_1(2020) = 200, u_1(2021) = 221, u_1(2022) = 198$, and $u_1(x) = 0$ for $x \notin \{2020, 2021, 2022\}$...
    - ... and we observe $y_1(2020) = 51, y_1(2021) = 51, y_1(2022) = 60$
    - $u_1(2020) = 136, u_2(2021) = 231, u_2(2022) = 90$, and $u_2(x) = 0$ for $x \notin \{2020, 2021, 2022\}$...
    - ... and we observe $y_2(2020) = 42, y_2(2021) = 37, y_2(2022) = 98$
    - $u_1, u_2 \in \mathcal{U},\ y_1, y_2 \in \mathcal{Y},\ (u_1, y_1), (u_2, y_2) \in \mathcal{S}$

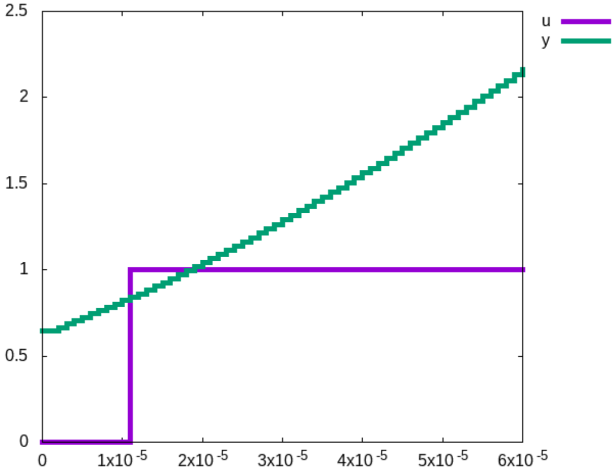
Example: determine the output voltage of a buck DC-DC converter



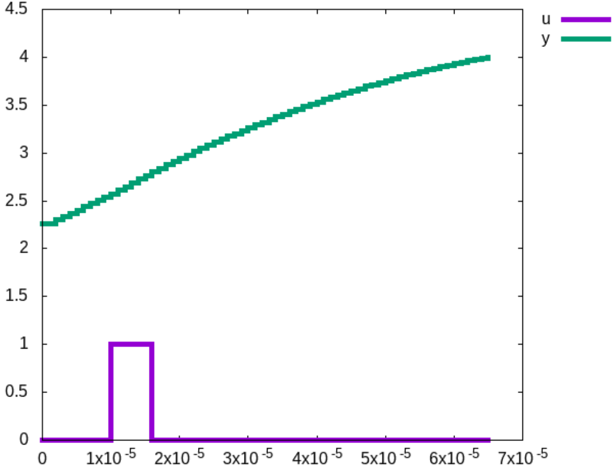
Some Background: Systems

- Example: determine the output voltage of a buck DC-DC converter
  - $T \subset \mathbb{R}$
  - $U \subset \{0, 1\} \times \mathbb{R}$
    - u may be closed (0) or open (1) at any time
    - V_i may be any real number
  - $Y \subset \mathbb{R}$: observed output voltage v_O
  - example of input-output (times are in microseconds):
    - $u_1(t) = (0, 5)$ for all $t \in [0, 10]$, $u_1(t) = (1, 5)$ for all $t \in [10, 100]$
    - $u_2(t) = (0, 15)$ for all $t \in [0, 9]$, $u_2(t) = (1, 10)$ for all $t \in (9, 15]$, $u_2(t) = (0, 7)$ for all $t > 15$

Result for u_1

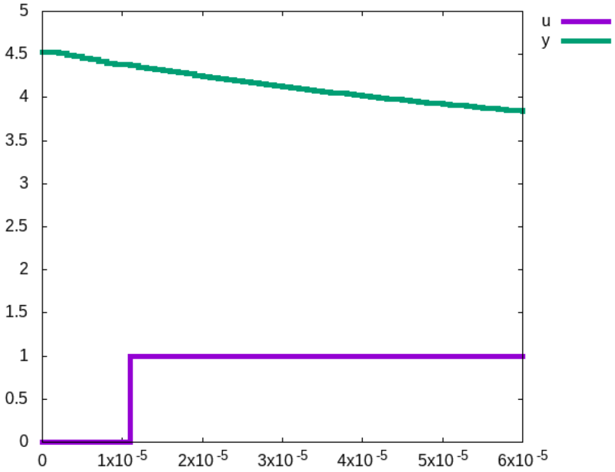


Result for u_2

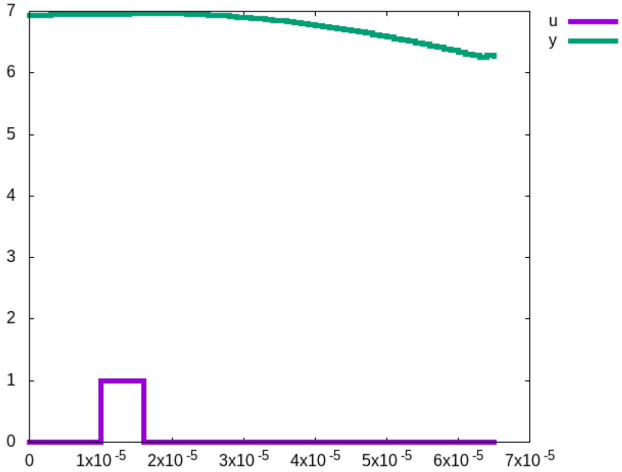


Some Background: Systems

... but this is also a result for u_1



... and this is also a result for u_2



Some Background: Systems

- Is this a non-deterministic system??? NO!
- The point is that output is not determined by input only
  - though for some systems this is the case: number of students above
- The missing element is the *state*
  - essentially, the input/output function has side effects...
- Thus, the output (for *deterministic* systems) is a function of both the input and the state
  - in the examples above, we made different choices for the starting state

Some Background: Systems

- For our purposes, a system will be defined by a 6-tuple $\mathcal{S} = \langle T, U, Y, X, \eta, \phi \rangle$:
  - U, Y are sets of possible input and output values, resp.
  - T is a set of times
    - if $T \subseteq \mathbb{R}$ then we have a *continuous-time* system
    - if $T \subseteq \mathbb{N}$ then we have a *discrete-time* system
  - X is a set of states
    - may be either finite or infinite
    - if $T \subset \mathbb{N}$ and $|X| < \infty$ then we essentially have a Kripke structure
  - $\eta : T \times X \times U \to Y$ defines the output function
  - $\phi : T \times T \times X \times \mathcal{U} \to X$ defines the state transition function
    - recall that $\mathcal{U} = \{f \mid f : T \to U\}$

Some Background: Systems

- $\eta : T \times X \times U \to Y$ is as expected
  - given the current time, the current state, and the current input, we can compute the output
- $\phi : T \times T \times X \times \mathcal{U} \to X$ is somewhat more complicated than expected
  - one would expect $\phi : T \times X \times U \to X$
  - actually, this is enough for most systems
- For some systems, the state transition function depend on some *sequence* of inputs, not only the last one
  - so we need a function, defined at least on an interval $[t_0, t)$
  - this is why ϕ also takes two times instead of one
- 3 conditions must hold for η and ϕ: *causality*, *consistency* and *separation*

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Some Background: Causal Systems

- $\forall t, t_0 \in T, x_0 \in X.\ (t \geq t_0 \wedge u|_{[t_0,t)]} = u'|_{[t_0,t)}) \Rightarrow \phi(t, t_0, x_0, u|_{[t_0,t)]}) = \phi(t, t_0, x_0, u'|_{[t_0,t)})$
- That it is, if we fix the first 3 arguments t, t_0, x_0 of ϕ...
- ... and we provide, as a fourth argument, two possible different functions u, u'...
- ... which however output the same values in the interval $[t_0, t)$...
- ... then the final value of ϕ does not change
- Thus, what happens in the interval $[t_0, t)$ *causes* the system to go to one single state

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Some Background: Consistent Systems

- $\forall t \in T, x_0 \in X, u \in \mathcal{U}.\ \phi(t, t, x_0, u) = x_0$
- Recall that, for a call $\phi(t, t_0, x, u)$, u is considered in the interval $[t_0, t)$
- Thus, in a call $\phi(t, t, x_0, u)$, we are considering the empty interval $[t, t)$
- Hence, we have no input at all!
- Of course, without inputs, the system cannot change its current state

Some Background: Separation Property of Systems

- $\forall t, t_0, t_1 \in T, x_0 \in X, u \in \mathcal{U}.\ (t > t_1 > t_0) \Rightarrow$
  $\phi(t, t_0, x_0, u|_{[t_0,t)]}) = \phi(t, t_1, \phi(t_1, t_0, x_0, u|_{[t_0,t_1)}), u|_{[t_1,t)})$
- In few words: the state you obtain if you go straight from t_0 to t, is the same state you would obtain if:
  - you first go from t_0 to some intermediate t_1
    - i.e., $x_1 = \phi(t_1, t_0, x_0, u|_{[t_0,t_1)})$
  - and then from t_1 to t
    - i.e., $\phi(t, t_1, x_1, u|_{[t_1,t)})$

Some Background: Hybrid Systems

- Note that the set of states X may be multi-dimensional
  - e.g., $X = \mathbb{R}^3$, or $X = \{1, 2, 3\} \times \mathbb{Z}$
- Thus, also ϕ may be multi-dimensional
- Informally, if X has dimension n, then we will have *n state variables*
  - recall that the same holds for U, Y: we may have multiple *input* and *output variables*
- Hybrid systems: those for which some variables are continuous and other are discrete
  - in some texts, a "hybrid system" have some variables depending on $T = \mathbb{N}$ and some other on $T = \mathbb{R}$
- This is exactly the case of cyber-physical systems!
  - plant + controller/monitor
  - plant is continuous, controller/monitor is discrete

Some Background: Special Systems

- With some semplification, a system is *time-invariant* iff $\forall t, t_0, t_1 \in T, x \in X, u \in \mathcal{U}.\ \phi(t, t_0, x, u) = \phi(t - t_0, 0, x, u) \wedge \eta(t, x, u(t)) = \eta(t_1, x, u(t_1))$
  - that is, the absolute time is not important
  - the *relative* time is
    - given a state x, system evolution from 1 to 3 seconds and from 10 to 12 seconds is the same
- For time-invariant systems, we can always set $t_0 = 0$
- For time-invariant systems, we can also write $\dot{x}(t) = \phi(x(t), u(t)), y(t) = \eta(x(t), u(t))$

Some Background: Special Systems

- With some semplification, a system is *linear* iff
  - U, Y, X are linear spaces
    - that is, any linear combination $\sum_{i=1}^{n} a_i x_i$ is in X etc
  - $\mathcal{U}$ is a linear subspace of $U^T = \{f \mid f : T \to U\}$
    - again, any linear combination $\sum_{i=1}^{n} a_i u_i(t)$ is in $\mathcal{U}$
  - fixed any 2 times $t, t_0 \in T$ as first 2 arguments, ϕ is linear in the remaining 2 arguments
    - $\phi(t, t_0, x, u) = A \cdot [x, u] + b$ for some A and b
    - A, b may depend on t, t_0, but not on x, u
  - fixed any time $t \in T$ as first argument, η is linear in the remaining 2 arguments
- Linear systems are easy to model, simulate and verify
- With some semplification, a system is:
  - a finite-state system if U, X, Y are finite sets (Kripke structure)
  - a finite-dimensional system if U, X, Y are linear finite-dimension spaces

Some Background: Generating Functions

- For discrete-time systems, we have that $x(t+1) = \phi(t+1, t, x(t), u|_{[t,t+1)}) = \phi(t+1, t, x(t), u(t)) = f(t, x(t), u(t))$
  - first and second argument of ϕ are not independent...
  - f has the same domain of η
- For continuous-time systems, we focus on *regular* systems, i.e., those systems for which ϕ is differentiable and there exists a function f s.t.
  - $\frac{d\phi(t,t_0,x,u)}{dt} = f(t, \phi(t, t_0, x, u), u(t))$
  - with the initial condition that exists an $x_0 \in X$ s.t. $x_0 = \phi(t_0, t_0, x_0, u)$
  - often, it is easier to provide f than ϕ
- Using Newtonian notation, we have $\dot{x}(t) = f(t, x(t), u(t))$
- For time-invariant systems, we have $\dot{x}(t) = f(x(t), u(t)), y(t) = \eta(x(t))$

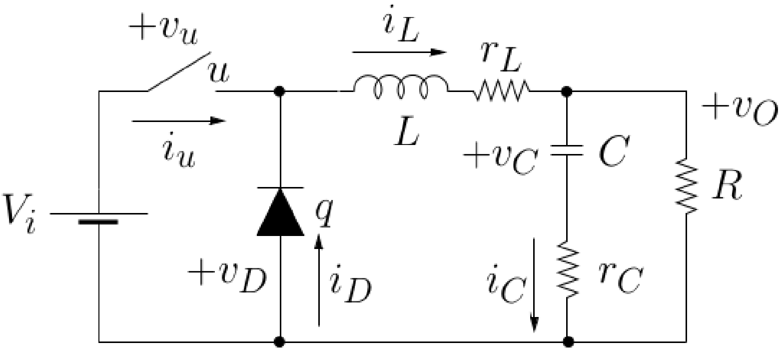
Some Background: System For Students' Example

- $X \subseteq \mathbb{N}^3$, $U, Y \subseteq \mathbb{N}$, $T = \mathbb{N}$
- Parameters $\alpha_i(t) \in [0, 1]$ is ratio of students passing an year t
- $x_1(t+1) = (1 - \alpha_1(t))x_1(t) + u(t)$
- $x_i(t+1) = (1 - \alpha_i(t))x_i(t) + \alpha_{i-1}(t)x_{i-1}(t)$ for $i = 2, 3$
- $y(t) = \alpha_3(t)x_3(t)$
- Note that, if $\alpha_i(t) = 1$ for all t, then states are not needed, as we have $y(t) = u(t-3)$
- Summing up:
  - $f(t) = \begin{pmatrix} (1 - \alpha_1(t))x_1(t) + u(t) \\ (1 - \alpha_2(t))x_2(t) + \alpha_1(t)x_1(t) \\ (1 - \alpha_3(t))x_3(t) + \alpha_2(t)x_2(t) \end{pmatrix}$

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Some Background: System For Buck DC/DC Converter



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Some Background: System For Buck DC/DC Converter

- $X \subseteq \mathbb{R}^2, U \in \{0,1\} \times \mathbb{R}, Y \subseteq \mathbb{R}, T = \mathbb{R}$
- $L, C, R, r_L, r_C \in \mathbb{R}$ are system parameters
  - we will also use 6 real numbers $a_{i,j}$ for $i \in \{1,2\}, j \in \{1,2,3\}$ which are functions of such parameters
  - e.g., $a_{2,3} = -\frac{1}{L}\frac{r_C R}{r_C + R}$
- Variables for state are $i_L, v_O, v_D, i_D, v_u, i_u$ (real) and q (boolean)
- Variables for input are u (boolean) and V_i (real)
- $y(t) = v_O(t)$, thus η is easy
- ϕ is defined by cases in the following slide
  - for the definition of ϕ, some other (*auxiliary*) variables are useful: v_D, i_D, v_u, i_u (real) and q (boolean)
  - $R_{on} \approx 0, R_{off} >> R_{on}$ are fixed parameters

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Some Background: System For Buck DC/DC Converter

We omit (t) for better readability
There must exist a value for $v_D, i_D, v_u, i_u \in \mathbb{R}, q \in \{0, 1\}$ s.t.

$$\dot{i_L} = a_{1,1} i_L + a_{1,2} v_O + a_{1,3} v_D \quad (1)$$
$$\dot{v_O} = a_{2,1} i_L + a_{2,2} v_O + a_{2,3} v_D \quad (2)$$

$$q \rightarrow v_D = R_{\mathrm{on}} i_D \quad (3)$$
$$q \rightarrow i_D \geq 0 \quad (4)$$
$$u \rightarrow v_u = R_{\mathrm{on}} i_u \quad (5)$$
$$v_D = v_u - V_{in} \quad (6)$$
$$\bar{q} \rightarrow v_D = R_{\mathrm{off}} i_D \quad (7)$$
$$\bar{q} \rightarrow v_D \leq 0 \quad (8)$$
$$\bar{u} \rightarrow v_u = R_{\mathrm{off}} i_u \quad (9)$$
$$i_D = i_L - i_u \quad (10)$$

Both ODEs and algebraic equations

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Modeling in Modelica

- Modelica is an open-source language for specifying (complex) systems
  - developed by experts starting in late 1990s
- Many implementations exist
  - OpenModelica+simForge, Dymola, Simulation X, MapleSim, MathModelica
  - here we will stick to OpenModelica+simForge
- Also see Modelica slides

Modeling in Modelica

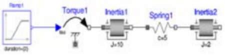
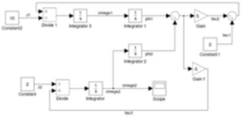
- Object-oriented language: classes and objects (i.e., class instances)
  - strongly typed
- Compositional modeling:
  - break up the system in subsystems (components)
  - connect the components
- Very useful for complex systems, with many components
  - some standard components already defined, e.g., resistors, flows etc
- May use equations, also with derivatives
- Generates a C program, thus it is very efficient

Modeling in Modelica

- Synchronous data flow principle: time is the same for all components
  - such as clocks for digital systems, but in Modelica it may be continuous
- May specify "algorithms" using assignments, ifs, whiles, etc
  - all variables must be instances of some class
  - this also includes integers and reals
- Acausal modeling: simply first provide the equations for each object, then connect the objects between them
  - other modeling languages, e.g., Simulink, requires to first design the full chain of connections...
  - ...and to make computation in sequence
  - Modelica allows both causal and acausal modeling
  - physical "reality" is lost
- Modelica easier for modelers, Simulink easier for computers

Acausal Modeling

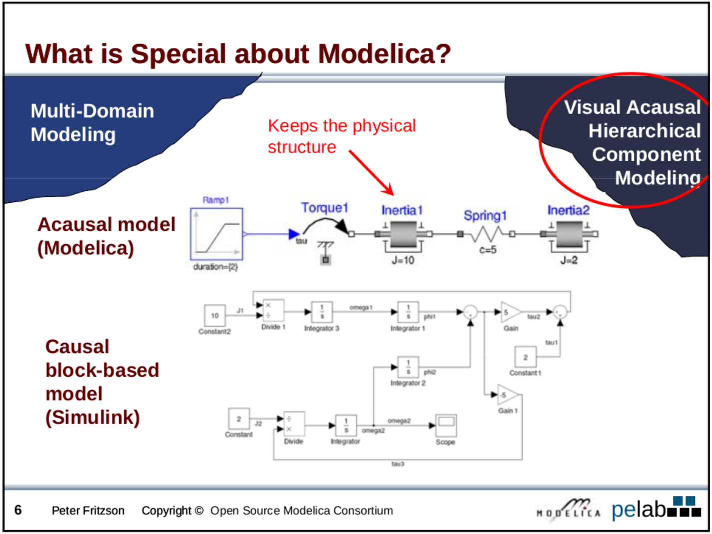
The order of computations is not decided at modeling time

	Acausal	**Causal**
Visual Component Level		

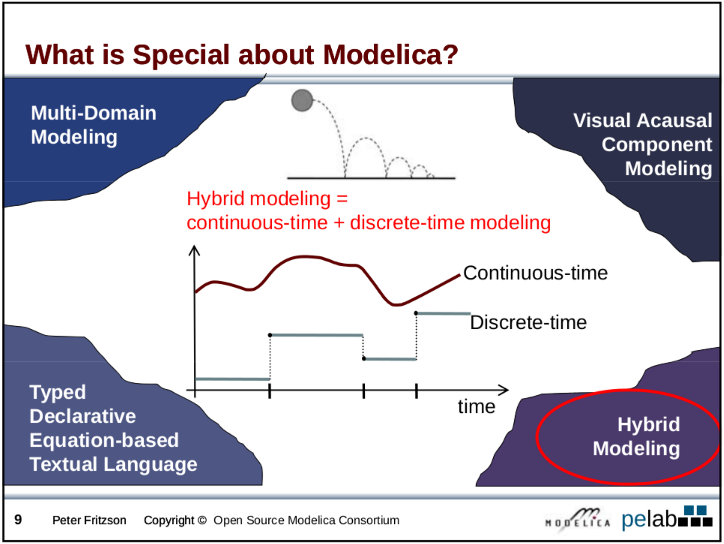
Equation Level	A resistor *equation*:
	`R*i = v;`
		Causal possibilities:
		`i := v/R;`
		`v := R*i;`
		`R := v/i;`

DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Modelica: Acausal Modeling



Modelica: Hybrid Modeling



Modelica: Toy Example

- Text file with `.mo` extension, let us say `model.mo`

```
class Example
  output Real x, y, z;
  algorithm
    when initial() then //at 0, both this...
      x := 0; // Pascal-like assignments
    elsewhen sample(0, 1) then // ... and this
      x := 1;
      y := pre(x); //0 till 1, then always 2
    elsewhen sample(0, 0.5) then
        //elsewhen order is important! from bottom to top
      x := 2;
      z := pre(x); //0 till 0.5, then always 1
    end when;
end Example;
```

Modelica: Toy Example

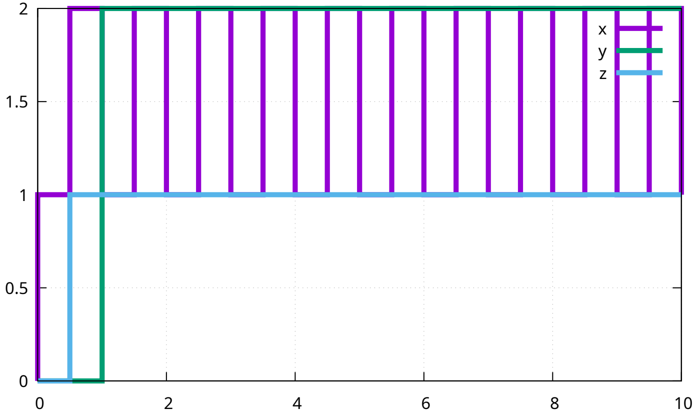
- Text file with `.mos` extension, let us say `run.mos`

```
loadModel(Modelica);
getErrorString(); // should be used after every command,
                  // skipped in the following
loadFile("model.mo");
//Example is defined in model.mo
simulate(Example, stopTime=10);
//x, y, z are variables of Example
plot({x, y, z}, externalWindow=true,
     fileName="Example_res.mat");
```

Modelica: Toy Example

- Run the command `omc run.mos`
  - of course, you must have installed `omc` for your OS
- This has the following effect:
  1. generates a C program `model.c`
  2. compiles `model.c` to obtain the executable file `model`
  3. executes `model`
  4. outputs both a file `Example_res.mat` and a graphical window with the graph of variables `x, y, z` as function of time

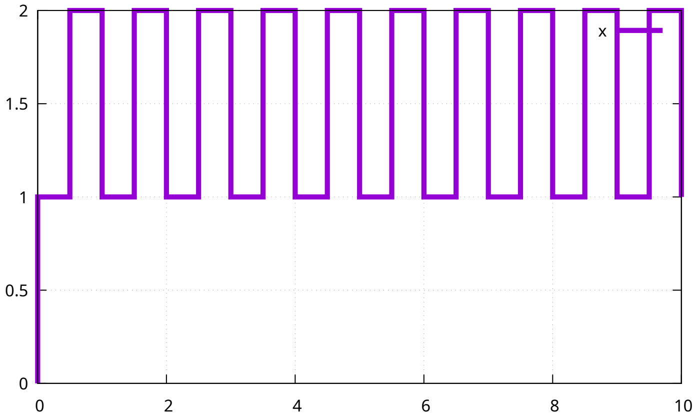
Modelica: Commands Chain



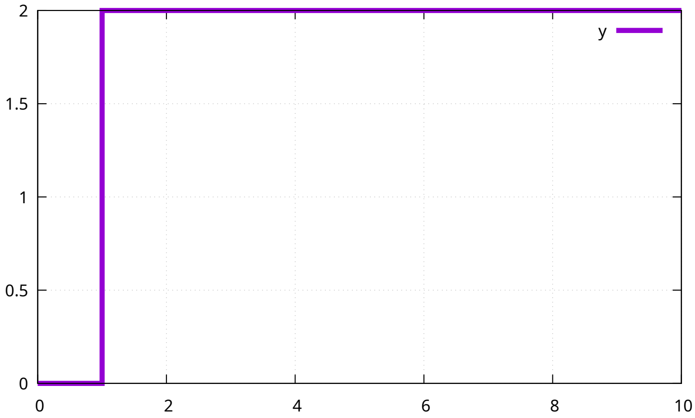
UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

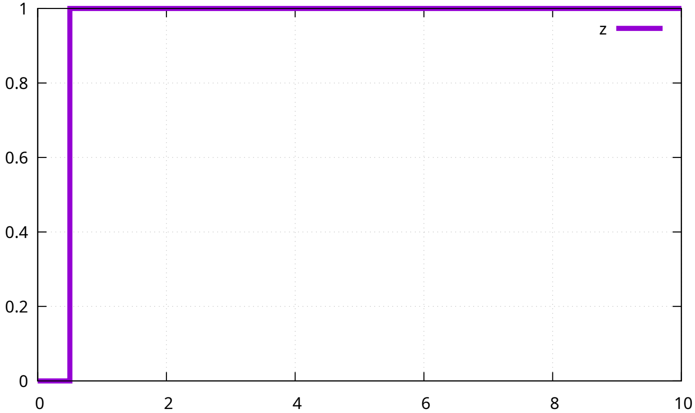
Modelica: Commands Chain



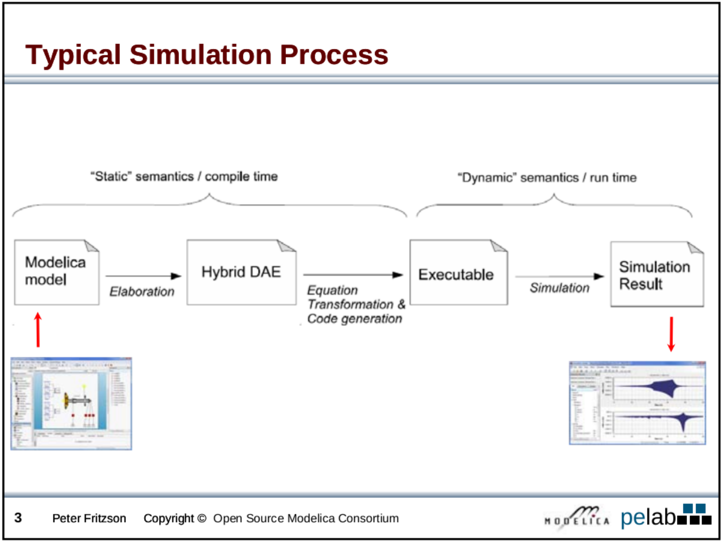
Modelica: Commands Chain



Modelica: Commands Chain



Modelica: Commands Chain



Modelica: Toy Example Examined

```
class Example
  output Real x, y, z;
  algorithm
    when initial() then //at 0, both this...
      x := 0;
    elsewhen sample(0, 1) then // ... and this
      x := 1;
      y := pre(x); //0 till 1, then always 2
    elsewhen sample(0, 0.5) then
      x := 2;
      z := pre(x); //0 till 0.5, then always 1
    end when;
end Example;
```

Modelica: Toy Example Examined

- `Example` is a class defined by the modeler: Modelica is OO
- It has 3 real-valued variables, which may become the input for other blocks
- The dynamics is an algorithm based on the `sample` construct
  - `when initial()` C executes code in C at time 0
  - `when sample(`A`,` B`)` C executes code in C every $A + Bx$ seconds, for $x \in \mathbb{N}$
  - there may be multiple `elsewhen sample(`A`,` B`)` C triggered at a given time
  - they are all executed, starting from the bottom of the file
  - now explain the output of the previous example...
- In expressions, `pre(var)` holds the value of `var` *before* the current event

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Modelica: Derivatives

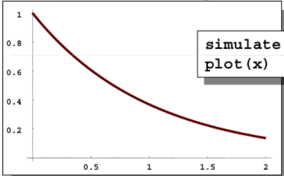
Simplest Model – Hello World!

A Modelica "Hello World" model

Equation: x' = - x
Initial condition: x(0) = 1

```
class HelloWorld "A simple equation"
  Real x(start=1);
equation
  der(x)= -x;
end HelloWorld;
```

Simulation in OpenModelica environment




```
simulate(HelloWorld, stopTime = 2)
plot(x)
```


DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

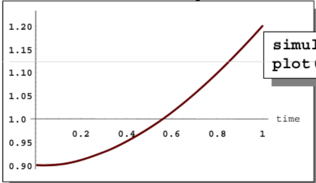
Model Including Algebraic Equations

Include algebraic equation

Algebraic equations contain no derivatives

```
class DAEexample
  Real x(start=0.9);
  Real y;
equation
  der(y)+(1+0.5*sin(y))*der(x)
    = sin(time);
  x - y = exp(-0.9*x)*cos(y);
end DAEexample;
```

Simulation in OpenModelica environment




```
simulate(DAEexample, stopTime = 1)
plot(x)
```


DISIM
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Modelica: Derivatives and Algebraic Equations

- `time` is a special variable, holding current simulation time
- System dynamics of previous example is defined as

$$\dot{y} + (1 + \frac{\sin y}{2})\dot{x} = \sin t$$

$$x - y = e^{-0.9x} \cos y$$

- Can be transformed in “normal” form by adding state variables:

$$\dot{x} = \frac{\sin t - y_1}{1 + \frac{\sin y}{2}}$$
$$\dot{y} = y_1$$
$$z_1 = e^{-0.9x}$$
$$z_2 = \cos y$$
$$x = z_1 z_2 + y$$
$$y = x - z_1 z_2$$

Modelica: Subsystems and Connections

- Till now, stand-alone systems with just one component
- Modelica allows compositional modeling of many components
- Each component is modeled autonomously, by simply looking at the interaction with the environment (input/output)
- Complex systems are made of *connected* components
- Connectors can be explicitly defined
  - causal: input/output relation is explicitly stated
  - acausal: input/output relation is left unspecified
  - Modelica will understand which is input and which is output

Modelica: Toy Example With 2 Components

Mind the difference between = and :=

```
model ContinuousBehav

  Boolean x;
  Real i (start = 1);

  equation
    (if x then 0.5*time else -0.1*time)*der(i) = time;
end ContinuousBehav;
```

Modelica: Toy Example With 2 Components

```
model GenerateBoolInputs

  Boolean x;
  parameter Real sampling = 1.0;

  algorithm
    when initial() then
      x := false;
    elsewhen sample(sampling, sampling) then
      x := not(x);
    end when;
end GenerateBoolInputs;
```

Modelica: Toy Example With 2 Components

```
model BoolCont

  GenerateBoolInputs gbi;
  ContinuousBehav cb;

  equation
    gbi.x = cb.x;

end BoolCont;
```

Modelica: Toy Example With 2 Components

May also collect parts of commands in a file `file.mos` and use `runScript("file.mos")`

```
loadModel(Modelica);
getErrorString();
loadFile("model.mo");
simulate(BoolCont, stopTime=10);
plot({gbi.x, cb.i}, externalWindow=true,
     fileName="BoolCont_res.mat");
```

Modelica: Passing of Time

- For all objects defined, the time passes in the same way
  - it is a kind of common clock, as in digital circuits
- This is of course consistent with physical reality
  - components are close enough...
- It is always continuous time, but using `sample` we can also have discrete time

Modelica: Algorithms and Equations

- Both may be used, the modeler has to choose
  - of course, `x := x + 1` inside an algorithm is ok, `x = x + 1` in an equation is not
  - using imperative vs. declarative style is left to the modeler
  - in some cases, algorithm is more natural, in some other equation has to be preferred
  - note that loops and ifs are available in both formats
  - e.g., `a = (if b then 1 else 2);` vs `if (b) then a:=1; else a:=2; end if;`
  - or simply `a := (if b then 1 else 2);`
- Algorithms, as well as equations solving, does not cause time to pass
  - number of computation steps required is not important

Modelica: Algorithms

- Generally speaking, `when` A B clauses triggers the corresponding block B when condition A is true
- A can be any boolean expression, not only `sample`
- Functions may also be defined and used
  - time does not pass during function calls
  - again, number of computation steps is not important
  - must have input and output
- External C or Fortran functions may be called

```
external "C" result = myfun();
annotation(Include = "#include \"myfile.c\"");
```

Modelica: Events

- Discrete events happens in a discrete number of time points
  - given that the simulation terminates somewhere, it is actually a finite number of points
- We saw `initial` and `sample`, there is also `terminal`
  - triggered at the end of the simulation
- Simulation ends either because of:
  - the `stopTime` attribute inside `simulate` command
  - a `terminate` statement

Modeling in Simulink

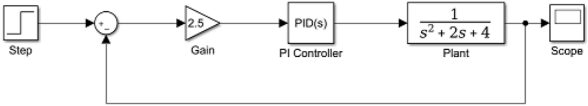
- Simulink is a graphical extension to MATLAB
  - MATLAB itself is proprietary, but UnivAQ provides it to students
- Main goal: modeling and simulation of systems
  - also non-linear ones
- Also see `https://ctms.engin.umich.edu/CTMS/index.php?aux=Basics_Simulink`
- No way of simply writing a text file: you have to use the GUI and manipulate graphical objects
  - model files are saved in a binary proprietary `SLX` format

Modeling in Simulink

- Two major classes of objects: blocks and lines
  - blocks used to generate, modify, combine, output, and display *signals*
  - lines used to connect blocks, i.e., transfer signals from one block to another
  - again, a common clock for all objects in a model
- Suppose you create a new or open an existing Simulink model file
- How to add a new block:
  - click “Library Browser”
  - select the type of block you need
    - hundreds of types available, could also be searched by name
  - drag it to the model window
  - by double clicking, you can edit the properties

Modeling in Simulink

- How to add a new connecting line:
  - simply drag the mouse from the first object to the second object
- If you are connecting an object with a line:
  - first make a dangling line from the destination
  - connect the end of such line with the "source" line
  - this will make the source line bifurcated



Modeling in Simulink

Most notable types of blocks:

- Sources: used to generate various signals
- Sinks: used to output or display signals
- Continuous: continuous-time system elements
  - transfer functions, state-space models, PID controllers, etc.
- Discrete: linear, discrete-time system elements
  - discrete transfer functions, discrete state-space models, etc.
- Math Operations: contains many common math operations
  - gain, sum, product, absolute value, etc.
- Ports & Subsystems: contains useful blocks to build a system