Hybrid acausal modeling using Modelica

Presentation of Modelica
Outline

Introduction to Acausal Modeling
  Definitions
  Comparison with causal modeling

Presentation of Modelica
  Basic language constructs
  Piecewise continuous-time semantics
  Discrete-time semantics
Physical System:
A physical system is an entity that can be separated from its environment by means of conceptual limits
A physical system interacts with its environment, which results in observable changes over the time

Physical Subsystem:
Physical systems are themselves composed of conceptual entities called subsystems
The hierarchy of physical subsystems is potentially infinite
Physical subsystems can be considered as physical systems whose environment is the union of the other subsystems and of the enclosing system's environment

Physical system point of view:
Observed changes are explained in terms of interactions between a system and its environment or in terms of interactions between subsystems
Introduction to Acausal Modeling: definitions

(Acausal) Model:
One way of studying physical systems is to build convenient abstractions of them called models.

A model is composed of:
• Variables
• Relations between variables

The variables in a model are functions of time to observable quantities (they implicitly expose changes inside models).
The relations in a model act as constraints between the values variables take at each instant.

Models interact through constraints between some of their variables.

Simulation:
A simulation consists in extracting information from a model one or more time instants.
Causal Modeling consider a rather different definition of "model":
A causal model is composed of:
- Inputs
- Outputs
- Variables
- State variables
- Relations between inputs and (state) variables constraining the value of outputs and variables
- Relations between inputs and (state) variables constraining the value of the derivatives of state variables

Inputs of causal models handle data coming from the environment
Outputs of causal models handle data to be exported to the environment

(State) variables of causal models are used to compute observable quantities

The key point is: data flow is explicit, i.e., it is possible to simulate a causal model using value propagation first and then integration
Introduction to Acausal Modeling: comparison with Causal Modeling

Acausal models are easy to build and modify
Acausal models require highly elaborated tools to handle them efficiently
Acausal modeling is a convenient way to express specifications

Causal models are difficult to build and extremely hard to modify
Causal models generally don't require elaborated tools to handle them efficiently
Causal modeling is a convenient way to express explicit computations
Presentation of Modelica: basic language constructs

Modelica is a modeling language that allows specification of acausal models. Modelica programs are organized as sets of classes which describe the elements used to build models. A class can be considered as the description of a family of objects (its instances) that share common properties. Modelica supports the following kinds of classes:

- Record
- Function
- Package
- Connector
- Model
- Block

The first three kinds of classes are common abstractions encountered in other general-purpose programming languages, while the other ones are Modelica-specific.
Modelica supports the following primitive datatypes:

- Booleans
- Enumerations
- Integers
- Reals
- Strings

Booleans include the two constants denoting "true" and "false"

Enumerations include families of enumerated types whose members are symbolic constants

Integers denote the (mathematical) set of integer numbers

Reals denote the set of (mathematical) real numbers, whose decimal members can be written literally in programs using C-like syntax

Strings denote the set of character strings

There is actually no datatype to represent "hardware-oriented" types such as 2-complement n-bits integers, n-bit floating-point numbers, etc.
Syntax of literal constants is rather classical:

**Boolean constants:**
- `true`
- `false`

**Integer constants:**
- `0`
- `-123456789012345678901234567890`

**Real (decimal) constants:**
- `3.1415927`
- `-1.2345e3`

**String constants:**
- "this is a string constant"
- "\tthis another\n\tstring constant"
Modelica supports structuration of data by means of:

- Records
- Arrays

Records are formed by aggregating objects of heterogeneous types, each object being associated with a unique key (field name).

The keys and the types of associated objects are shared among all instances of a given record class.

Arrays denote homogeneous multidimensional arrays.

Arrays and records can be used in combination to build more elaborated data structures.

Sum types and general recursive types are not currently possible in Modelica, which means that data structures such as trees of arbitrary depth are not directly supported by the language.
There is no dedicated syntax to denote structured literal constants; one has to make use of constructors or predefined operators to build them:

**Record construction:**
- `Complex(re=0, im=1)`
- `Point3D(x=1, y=2, z=3)`

**Array construction:**
- `{ 1, 2, 3 }`
- `{{ 1, 2 }, { 3, 4 }}`
- `zeros(2, 3)`
- `fill(3.14, 2, 3, 4)`
Definitions of classes in Modelica obey to homogenous syntactic rules:

\[
\text{class\_definition ::= class\_kind class\_name class\_contents \"end\" class\_name ;;
}\]

\[
\text{class\_kind ::= \"record\" | \"function\" | \"package\" | \"connector\" | \"model\" | \"block\"
}\]

\[
\text{class\_comment ::= \"\\\" comment \"\\\"
}\]

The content of a class definition depends of course of the kind of class being actually defined: record classes contain definitions of fields, function classes contain definitions of input and output formal parameters, internal variables as long as executable statements, etc.

The content of a class definition can include other class definitions that may refer to elements in the parent class; the usual static name lookup rules found in statically scoped languages apply

Despite syntactic similarities, some kinds of classes represent completely different kinds of entities, semantically speaking
Definitions of records include subdefinitions of:

Eventually, structural parameters used to denote array dimensions

Fields, used to hold information

```modelica
record Complex
  Real re;
  Real im;
end Complex;

record Data
  parameter Integer m, n;
  Real x[m, n];
  Real[m, n] y;
  Real[n] z[m];
end Data;
```
Definitions of functions include subdefinitions of:

- Eventually, some structural parameters used to denote array dimensions
- Formal parameters
- Eventually, some internal variables
-Executable statements (or call to foreign function, written in C for instance)
  - Executable statements include assignments, "while" loops, "for" loops and return

```modelica
function F
  input Real u1, u2;
  output Real y1, y2;
protected
  Real x1, x2;
algorithm
  x1 := u1 + u2;
  x2 := u1 - u2;
  y1 := x1 * x2;
  y2 := x1 / x2;
end F;
```

Definitions of packages include subdefinitions of:
Eventually, some constants
Classes

package P
    constant Real pi = 3.14;

    record R
        Real x;
    end R;

    function F
        input Real u;
        output Real y;
        algorithm
            y := 2 * pi * u;
    end F;

end P;
Modelica models are basically built of **connectors**, **variables** and **constraints**

Connectors can be seen as "communication channels" with the outside world

They form a convenient way to make models interact through external constraints (called **connection constraints**)  

They allow models to expose information about their internals to the outer world  

They are made of "connection variables" that belong to one of two categories:  

- Flow variables  
- Potential variables  

Connection constraints can be added to connection variables using the keywords "input" and "output"

```modelica
connector Pin  
  Real v;  
  flow Real i;  
end Pin;  
```
Variables are used by Modelica models to hold observable information.

They belong to one of the following categories:

- Constants
- Parameters
- "True" variables

The category a variable belongs to often determines when a Modelica tool computes its value:

- Constants and "structural parameters" are computed at instantiation time
- "Physical parameters" are usually computed (often given values) at instantiation time, but in some circumstances they have to be solved at initialization time
- "True" variables are usually computed at run time (except when the compiler manages to solve them earlier)

Values of variables are determined from the constraints that apply to them.

The way a Modelica compiler solves constraints to compute values of variables is implementation-dependant.
Modelica can describe hybrid models, i.e., models composed of a mix of discrete variables and piecewise-continuous variables.

Discrete variables that take "Real" values have to be declared explicitly as discrete to avoid confusion with piecewise-continuous ones (which is the default kind).

Variables taking boolean, integer, enumeration or string values are always discrete.

```model M
  Integer i;
  discrete Integer j;
  Real  x[10];
  discrete Real  y[2, 3];
...
end M;
```

Blocks are restricted models that only expose connectors whose connection variables are either tagged "input" or "output"

```block B
  input Real x;
  output Real y;
...
end B;
```
Modelica supports three kinds of constraints:

- **Instantaneous constraints**, which determine discontinuous evolution of discrete variables and of some piecewise continuous variables called *state variables*
  - they are only active at *event instants*
- **"Continuous" constraints**, which determine continuous evolution of piecewise continuous variables
  - they are always active
- **Connection constraints**, which permit models to interact through connector variables

Instantaneous constraints and "continuous" constraints can be expressed using either *equations* or *algorithms* in *equation sections* and *algorithm sections*, respectively

- Equations are either equalities between expressions or conditional equations (i.e., "if equations")
- Algorithms are sequences of statements corresponding to the common notion of "statement" encountered in imperative languages (assignments, "while" loops, "for" loops)

Connection constraints are expressed as special equations

Connections define implicit equations constraining variables of two or more models
"Continuous" constraints expressed as equations define differential equations that are parts of a bigger system of equations to be fulfilled during simulation:

\[
\text{equation} \\
\text{if } \text{time} > 10 \text{ then } \text{der}(x) = x; \text{ else } \text{der}(x) = -x; \text{ end if;}
\]
\[
\text{der}(y) = \text{if } \text{time} > 10 \text{ then } y \text{ else } -y;
\]
\[
a = b;
\]
\[
f(b) = 0;
\]
... 

"Continuous" constraints expressed as algorithms conceptually define equations computing variables that are assigned inside their body:

\[
\text{algorithm} \\
\text{if } \text{time} > 10 \text{ der}(x) := x; \text{ else der}(x) := -x; \text{ end if;}
\]
\[
\text{der}(y) := \text{if } \text{time} > 10 \text{ then } y \text{ else } -y;
\]
\[
\text{if } a > b \text{ then } \text{tmp} := a; a := b; b := \text{tmp}; \text{ end if;}
\]
\[
\text{while } a < b \text{ loop}
\]
\[
\text{tmp} := (a + b) / 2;
\]
\[
\text{if } f(\text{tmp}) < 0 \text{ then } a := \text{tmp}; \text{ else } b := \text{tmp}; \text{ endif;}
\]
\[
\text{end while;}
\]
...
"Continuous" constraints often appear in variable declarations under the form of *declaration equations*:

```model M
  parameter Real a := 1;
  Real x = y + 1;
  Real y = a * x;
end M;
```

The above model is equivalent to the following one:

```model M
  parameter Real a;
  Real x;
  Real y;
  algorithm
    a := 1;
  equation
    x = y + 1;
    y = a * x;
end M;
```
A high-level model (i.e., built as an aggregation of submodels) can make usage of *modification equations* to facilitate hierarchical modification of default declaration equations:

```model Circuit
    Ground g;
    VoltageSource s(V=5);
    Resistor r(R=5000);

    equation
        connect(g.p, s.n);
        connect(s.p, r.n);
        connect(r.p, g.p);

    end Circuit;
```
Instantaneous constraints are written using the "when" syntactic construct:

```
when activation_condition_1 then
  equations_or_algorithms_1
else when activation_condition_2 then
  equations_or_algorithms_2
...
else when activation_condition_n then
  equations_or_algorithms_n
end when;
```

Whenever one or several activation conditions of an instantaneous equation become true during simulation, constraints associated to the first of them become instantaneously active.
Instantaneous constraints are used to update discrete variables and continuous state variables at time instants:

```
when time > 10 then
  reinit(x, 0);
  a = pre(a) + 1;
else when pre(c) > pre(d) then
  c = pre(c) + pre(d);
  c + d = 0;
end when;

when x < y then
  x := 0;
  y := pre(y) + 1;
end when;
```
Connection constraints are written using the "connect" syntactic construct:

\[ \text{connect (m}_1\.p, \text{ m}_2\.n); \]
\[ \text{connect (m}_1\.p, \text{ m}_3\.n); \]

is equivalent to:

\[ \text{connect (m}_2\.n, \text{ m}_1\.p); \]
\[ \text{connect (m}_3\.n, \text{ m}_2\.n); \]

Connected elements should denote connectors of the same type

Connections in a model define a connection graph whose nodes are connectors

- Connected components of a connection graph are called connection sets
- Potential variables of connection sets are equal
- Flow variables of connection sets are summed to zero

Connections generally appear in "higher-level models", i.e., models build from ready-to-use submodels find in dedicated packages
Equivalent Modelica models can be defined using either equations, algorithms or a mix of the two:

```model Resistor1
    Pin p, n;
    Real v, i;
    parameter Real R;
    equation
        p.v - n.v = v;
        p.i + n.i = 0;
        i = p.i;
        v = R * i;
    end Resistor1;
```

```model Resistor2
    Pin p, n;
    Real v, i;
    parameter Real R;
    algorithm
        i := p.i;
        v := R * i;
        n.v := p.v + v;
        n.i := -p.i;
    end Resistor2;
```

```model Resistor3
    Pin p, n;
    Real v, i;
    parameter Real R;
    equation
        p.v - n.v = v;
        p.i + n.i = 0;
    algorithm
        i := p.i;
        v := R * i;
    end Resistor3;
```
A complete Modelica model (i.e., not a submodel) has to fulfill the *single assignment rule* in order to be considered correct:

- The number of conceptual scalar equations should match the number of variables
  - This applies to constants, parameters and variables
- The number of conceptual scalar equations in each branch of a conceptual conditional equation should be the same

```model Circuit
  Ground g;
  VoltageSource s(V=5);
  Resistor r(R=5000);
  equation
    connect(g.p, s.n);
    connect(s.p, r.n);
    connect(r.p, g.p);
  end Circuit;
```
Modelica model containing only "continous" constraints define a DAE system of the form:

\[ f(y', y, t) = 0 \]

One of the tasks of a Modelica processing tool is to process such models such that the result can be solved using conventional numerical solvers (DASSL, LSODAR, etc.)

A variable that appear as argument of the pseudo-function "der" do not necessarily denote a state variable

Determination of the "real" state variables of a system of equations is necessary in case of high-index system; this operation can lead to dynamic selections (i.e., made during simulation)

Detecting high-index systems is not always possible in practice; Modelica-based tools rely on a combination of symbolic and structural methods in order to give diagnostics

At at given moment during simulation, variables that are not selected as state variables are called algebraic variables
Presentation of Modelica: piecewise continuous-time semantics

During the simulation, the set of constraints to solve may change according to the value of some boolean condition (conditional systems)

The change of a boolean condition is called an event

*Event detection* is mandatory in Modelica-based tools

- The tool has to locate instants at which events are fired (using prediction methods or iterative methods)
- A starting point that satisfies the new set of constraints is determined at each event instant and the simulation is restarted

A Modelica-based tool automatically detects expressions that trigger events in models (typically, conditionals)

Of course, with no information, a tool may be too pessimistic, which results in slow simulations

Modelica features two pseudo-functions to help tools to perform efficient simulations by indicating information about smoothness of conditional equations or by disabling events explicitly

\[ x = \text{smooth}(1, \text{if } y > 0 \text{ then } y^2 \text{ else } -y^2); \]
\[ x = \text{noEvent}(\text{if } y \geq 0 \text{ then } \sqrt{y} \text{ else } 0); \]
Modelica is based on the *synchronous dataflow principle*:

- Discrete variables keep their values between two event instants
- Instantaneous constraints have to be fulfilled concurrently
- Computation or communication at event instants do not take time from the simulated system point of view
  - If computation or communication should be simulated, they have to be explicitly programmed

Modelica does not guaranty synchronization of events

Synchronization has to be programmed explicitly

```plaintext
when sample(0, 1) then
  ticks = mod(pre(ticks) + 1, 10);
  ...
end when;
when ticks == 0 then
  ...
end when;
```
Example of hybrid system

```model M
  Real x;
  discrete Real old_x, new_x;
  Integer count;
  equation
    f(der(x), x) = 0;
    when g(x) then
      reinit(x, 1);
    end when;
    when sample(0, 1e-3) then
      old_x = pre(new_x);
      new_x = x;
    end when;
    when old_x <= 0 and new_x > 0 then
      count = pre(count) + 1;
    end when;
  end
end M;```
Thank You