SNE SIMULATION NEWS EUROPE

Special Issue:

Object-oriented and Structural-dynamic Modeling and Simulation I

Volume 17 Number 2 September 2007, ISSN 0929-2268

Journal on Developments and Trends in Modelling and Simulation

Special Issue

ARGESIM

Editorial SNE Special Issue

Object-oriented and Structural-dynamic Modelling and Simulation I

The SNE special issues on *Object-oriented and Structural-dynamic Modelling and Simulation* emphasize on recent developments in languages and tools for object-oriented modelling of complex systems and on approaches, languages and tools for structural-dynamic systems.

Computer aided modelling and simulation of complex systems, using components from multiple application domains, have in recent years witnessed a significant growth of interest. In the last decade, novel equation-based object-oriented (EOO) modelling languages, (e.g. Modelica, gPROMS, and VHDL-AMS) based on acausal modelling using equations have appeared. These languages allow modelling of complex systems covering multiple application domains at a high level of abstraction with reusable model components.

This need and interest in EOO languages additionally raised the question for modelling approaches and language concepts for structural dynamic systems. Appropriate control structures like state charts in EOO languages also allow composition of model components 'in serial' – an interesting new strategy for modelling structural- dynamic systems.

There exist several different communities dealing with both subjects, growing out of different application areas. Efforts for bringing together these disparate communities resulted in a new workshop series, EOOLT workshop series, and established special sessions on structural-dynamic modelling and simulation (SDMS) within simulation conferences. In August 2007, the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools – EOOLT 2007 – took place in Berlin, with thirteen papers, and a special session at EUROSIM 2007 Congress (September 2007, Ljubljana) with seven papers concentrated on structural dynamic modelling (EUROSIM 2007- SDMS Special Session).

This SNE special issue on *Object-oriented and Structural-dynamic Modelling and Simulation – I* presents selected contributions from both events, presenting overview, state-of-the-art and development in the investigated subjects (five contributions from EOOLT, and two contributions from EUROSIM – SDMS).

Clearly, Modelica, the new standard for object-oriented and component-based physical modelling, plays an important role in many contributions. The first two contributions deal with structural concepts in MODELICA using UML - 'The Use of the UML within the Modeling Process of Modelica Models' by Ch. Nytsch-Geusen, and 'Towards Unified System Modeling with the Modelica ML UML Profile' by A. Pop et al. The third paper 'Hybrid Dynamics in Modelica: Should all Events be Considered Synchronous' by R. Nikoukhah raises problems with hybrid and structural-dynamic systems in Modelica and discusses general approaches to state event handling.

The fourth and the fifth paper, 'Enhancing Modelica towards variable structure systems' by D. Zimmer, and 'Functional Hybrid Modeling from an Object-Oriented Perspective' by H. Nilsson et al, link ideas of component-based object-oriented physical modelling and structural-dynamic modelling.

The sixth contribution 'Structure of Simulators for Hybrid and Structural-dynamic Systems' by N. Popper et al., reviews features of simulators for structural-dynamic systems and introduces different classes of state events. The last paper 'Modeling Structural Dynamics Systems in Modelica / Dymola, Modelica /Mosilab, and AnyLogic' by G. Zauner et al, presents features for hybrid and structural-dynamic modelling in equations-based object-oriented simulation languages.

Four contributions from EUROSIM SDS will be published together with EOOLT 2008 contributions in an SNE Special Issue '*Object-oriented and Structural-dynamic Modelling and Simulation – II* in 2008.

The editors would like to thank all authors for their co-operation and for their efforts, e.g. for sending revised versions, and hope that the selected papers present a good overview and state-of-the-art in object-oriented and structural-dynamic modelling and simulation.

Peter Fritzson, Linköping University, Sweden
François Cellier, ETH Zurich, Switzerland
Christoph Nytsch-Geusen, University of Fine Arts, Berlin, Germany
Peter Schwarz, Fraunhofer EAS – Dresden, Germany
Felix Breitenecker, Vienna Univ. of Technology, Austria
Borut Zupancic, Univ. Ljubljana, Slovenia

Proceedings EUROSIM 2007 - 6th EUROSIM Congress on Modeling and Simulation, B. Zupancic, R. Karba, S. Blazic (Eds.); ARGESIM / ASIM, Vienna (2007), ISBN: 978-3-901608-32-2;

Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools – EOOLT 2007, P. Fritzson, F. Cellier, Ch. Nytsch-Geusen (eds), Linköping University Electronic Press 2007, ISSN (online): 1650-3740; www.ep.liu.se/ecp/024/

Contents

SNE 17/2, September 2007

The Use of UML within the Modeling Process of Modelica Models

Christoph Nytsch-Geusen, Fraunhofer Institute, Germany, *christoph.nytsch@first.fraunhofer.de*

This paper presents the use of the Unified Modeling Language (UML) in the context of object-oriented modelling and simulation of hybrid systems with Modelica. The definition of a specialized version of UML for the graphical description and model based development of hybrid systems in Modelica—the UML^H—was taken place in the GENSIM project [1, 2]. For a better support of the modelling process, an UML^H editor with different views (class diagrams, statechart diagrams, collaboration diagrams) was implemented as a part of the Modelica simulation tool MOSILAB [3]. In the EOOLT-workshop the use of UML^H and its semantics will be demonstrated by the development of a simplified model of a Pool-Billiard game in Modelica.

Introduction

On the one hand, the Unified Modeling language (UML) is the established standard for the development and graphical description of object-oriented software systems [4]. UML offers a couple of diagrams, which describe different views (e.g. class diagrams, statechart diagrams, collaboration diagrams) on object-oriented classes. On the other hand *Modelica* [5] is a pure textual simulation language, which means the program code of long and highly structured models might be often heavy to understand. Thus, the combination of UML and Modelica was taken place within the GENSIM project. An UML editor for the Modelica based simulation tool MOSILAB was developed, which can be used for describing and generating Modelica models in a graphical way [3].

In this paper a special forming of UML for the modelling process of hybrid systems, the UML^H, will be presented. In a first step, the elements of the UML^H and their semantics for the Modelica-language will be introduced. After that, the use of UML^H will be illustrated by the example of a simplified version of a Pool-Billiard game.

1 UML^H and Modelica

The development of the UML^H was motivated by the following main reasons:

- to support the user within the modelling process of complex Modelica models in a easy manner,
- to have a method for the graphical documentation of the object-oriented construction of Modelica-models,
- to have a graphical analogy for the statechart extension of Modelica, which was introduced in the GENSIM project as a linguistic means of expression for model structural dynamics.

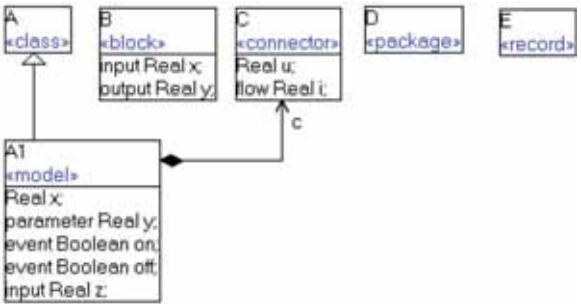


Figure 1. UML^H class diagram

The UML^H includes only a subset of the UML standard, which is necessary for the graphical description of Modelica models: the class diagram view, the statechart diagram view and the collaboration diagram view.

1.1 Class diagrams

A class diagram in UML^H is a rectangle, which contains in the upper part the class name and the Modelica class type. The optional lower part comprises the attributes (parameters, variables etc.) of the Modelica class. Inheritance and composition is expressed in the same way as in UML (compare with Fig. 1.)

Starting from this graphical notation, the correspondent Modelica code can be generated automatically, e.g. with MOSILAB (here, the UML^H diagrams are directly integrated within the Modelica code by the use of specialized annotations). The following code shows the classes `A`, `A1` and `C`, which are inner classes of the package `UML_H`:

```
1 package UML_H
2   annotation(UMLH(
      ClassDiagram="<umlhclass><name>…"));
3 class A
4    annotation(UMLH(classPos=[31,53]));
5 end A;
```

```
model A1
   annotation(
     Icon(Text(extent=…,string="A1", …));
   annotation(UMLH(classPos=[31,146]));
   extends A;
   event Boolean on;
   event Boolean off;
   Real x;
   input Real z;
   parameter Real y;
   C c;
   ...
end A1;
connector C
   annotation(UMLH(classPos=[192,54]));
   Real u;
   flow Real i;
end C;
...
end UML_H;
```

1.2 Collaboration diagrams

Collaboration diagrams in UML^H are also rectangles, which contain the object name and the type or the icon of the Modelica class, divided by a horizontal line. Four different connections types exist between the objects (see with Fig. 2.):

- Type 1: connections of connector variables (thin black line with filled squares at the ends)
- Type 2: connections of scalar variables (thin blue line with unfilled squares at the ends)
- Type 3: connections of scalar input/output variables (thin blue line with an arrow and a unfilled square)
- Type 4: multi-connections as a mixture of different connection types, e.g. type 1 and type 2 (fat blue line)

The following Modelica code expresses the collaboration-diagram of Fig. 2:

```
model System
   annotation(CompConnectors(
      CompConn(label="label2",
      points=[-81,52; -81,43;
              -24,43; 24,51])));
```

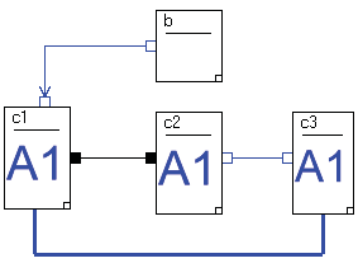


Figure 2. UML^H collaboration diagram

```
UML_H.A1 c1
   annotation(extent=[-87,72; -74,52]);
UML_H.A1 c2 annotation(extent=[…]);
UML_H.A1 c3 annotation(extent=[…]);
UML_H.B  b  annotation(extent=[…]);
equation
   // connection type 1
   connect(c1.c,c2.c)
     annotation(points=[-74,62;-57,62]);

   // connection type 2
   c2.y=c3.y annotation(points=[…]);

   // connection type 4 (mixture of type 1 and 2):
   connect(c1.c,c3.c)
       annotation(label="label2");
   c1.x=c3.x
       annotation(label="label2");

   // connection type 3:
   b.y=c1.z  annotation(points=[…]);
end System;
```

1.3 State chart diagrams

A statechart diagram in UML^H is represented as a rectangle with round corners. In general, a statechart diagram contains several states and the transition definition between the states. Figure 3 shows four different types of States:

- Initial states, symbolized with a filled circle,
- Final states, symbolized with a point in a unfilled circle,
- Atomic states, with a flat internal structure,
- Normal states, which can contain additional entry or exit actions and can be substructured in further statechart diagrams.

The transitions between the states are specified with an optional label, an event, an optional guard and the action part. The following code shows the corresponding code of the statechart section of the model `A1` (The new introduced statechart section is part of the Modelica language extension for model structural dynamics [6]):

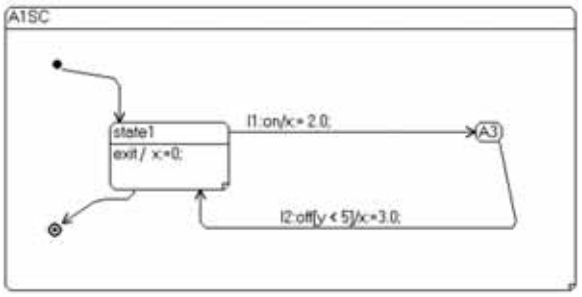


Figure 3. UML^H statechart diagram

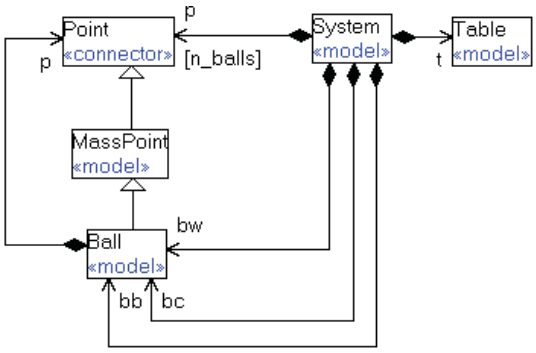


Figure 4. UML^{H} class diagram of the Pool-Billard model

```
1  model A1
2  ...
3  statechart
4  state A1SC extends State
5    annotation(extent=[-88,86; 32,27]);
6    state State1
7      extends State;
8      exit action x:=0; end exit;
9    end State1;
10   State1 state1
       annotation(extent=[-66,62;-41,48]);
11   State A3 annotation(extent=...);
12   State I5(isInitial=true)...;
13   State F7(isFinal=true)...;
14   transition I5->state1 end transition
15     annotation(points=[-76,73;-64,71;…]);
16   transition l1:state1->A3
       event on action x:= 2.0;
17   end transition annotation(points=...);
18   transition l2:A3->state1
       event off guard y < 5 action x:=3.0;
19   end transition ...;
20   transition state1->F7 end transition
       annotation...;
21 end A1SC;
22 end A1;
```

2 Example for UML^{H} modeling

The modelling and simulation of a simplified Pool-Billiard game shall demonstrate the advantages of the graphical modelling with UML^{H}.

2.1 Model of a Pool-Billard game

The system model of the Pool-Billiard game includes sub models for the balls and the table. The configuration of the system model is illustrated in Fig. 4. Following simplifications are assumed in the model:

- The Pool-Billiard game knows only a black, a white and a coloured ball.
- The table has only one hole instead of 6 holes.
- The collision-model is strong simplified.

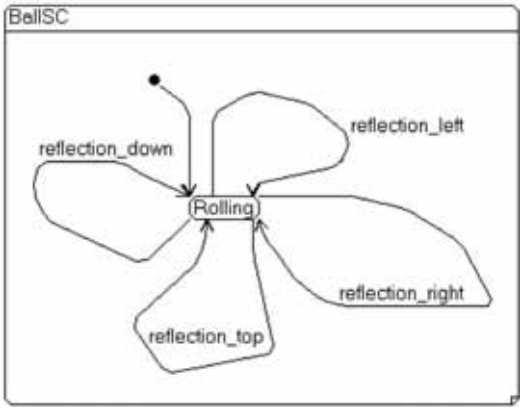


Figure 5. UML^{H} statechart diagram of the ball model

- The balls are moving between the collisions and reflections only on straight directions in the dimension x and y.
- The reflections on the borders take place ideal without any friction losses.
- The rolling balls are slowed down with a linear friction coefficient f_r:

$$m \cdot \frac{dv_x}{dt} = -v_x \cdot f_r, \quad \frac{dx}{dt} = v_x \tag{1}$$

$$m \cdot \frac{dv_x}{dt} = -v_x \cdot f_r, \quad \frac{dy}{dt} = v_y \tag{2}$$

Fig. 5 shows the statechart diagram for the ball model. After the model enter the state *Rolling*, the ball knows four reflection events, for the four different borders of the billiard table. Depending from the border event, the new initial conditions (velocity and position) after the reflections are set and the ball enters again the state *Rolling*:

```
1  model Ball
2    extends MassPoint(m=0.2);
3    parameter SIunits.Length width;
4    parameter SIunits.Length length;
5    parameter SIunits.Length d = 0.0572
         "diameter";
6    parameter Real f_r = 0.1
         "friction coefficient";
7    SIunits.Velocity v_x, v_y;
8    event Boolean reflection_left
         (start = false);
     ...
9    equation
10     reflection_left = if x < d/2.0;
11     m * der(v_x) = - v_x * f_r;
12     der(x) = v_x;
     ...
13   statechart
14     state BallSC extends State;
```

```
15        State Rolling;
16        State startState(isInitial=true);
          ...
17        transition startState -> Rolling
            end transition;
          ...
18        transition Rolling->Rolling
            event reflection_left
19          action v_x := -v_x; x := d/2.0;
20        end transition;
21        ...
22      end BallSC;
23  end Ball;
```

On the system level two different states (`Playing` and `GameOver`) and two types of events - the collision of two balls and the disappearance of a ball in the hole (compare with Fig. 6 and the program code) exist. If the white ball enters the hole, the game will be continued with the white ball from the starting point (transition from `Playing` to `Playing`). If the black ball disappears in the hole, the statechart is triggered to the state `GameOver`. If the coloured ball disappeared, the game is reduced for one ball - `remove(bc)` - and the numerical calculation will be continued with a smaller equation system (This model reduction mechanism takes place by using the model structural dynamics from MOSILAB [1]):

```
1  model System
2    parameter SIunits.Length d_balls = 0.0572;
3    parameter SIunits.Length d_holes = 0.15;
4    dynamic Ball bw, bb, bc;
                  // structural dynamic submodels
5    Table t(width = 1.27, length = 2.54);
6    event Boolean disappear_bw(start = false);
7    event Boolean disappear_bb(start = false);
8    event Boolean disappear_bc(start = false);
9    event Boolean collision_bw_bb(start =false);
     ...
10   event Boolean push(start = false);
11 equation
```

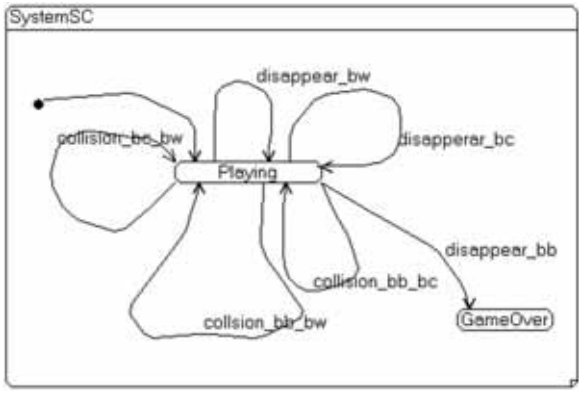


Figure 6. UML^H statechart diagram for the model

```
12   push = if fabs(bw.v_x)<0.005
                and fabs(bw.v_y)<0.005;
13   disappear_bw = if((p[1].x-0)^2
           + (p[1].y-0)^2)^0.5 < d_holes;
14   collision_bw_bb = if((p[2].x-p[1].x)^2
           + (p[2].y-p[1].y)^2)^0.5 < d_balls;
     ...
15 statechart
16 state SystemSC extends State;
17   State Playing, startState(isInitial=true),
           GameOver;
     ...
18   transition startState -> Playing action
19     bw := new Ball(d = d_balls,...); add(bw);
20     bb := new Ball(...); add(bb);
21     bc := new Ball(...); add(bc);
22   end transition;
23   transition Playing->Playing
           event disappear_bw action
       ...
24     remove(bw);
25     bw := new Ball(x(start=1.27/2.9),
                      y(start=0.6));
26   end transition;
27   transition Playing->Playing
           event disappear_bc action
28     ...
29     remove(bb);
30   end transition;
31   transition Playing->GameOver
           event disappear_bb end transition;
32   transition Playing->Playing
           event collision_bw_bb action
33     v_x := bw.v_x; v_y := bw.v_y;
34     bw.v_x := bb.v_x; bw.v_y := bb.v_y;
35     bb.v_x := v_x; bb.v_y := v_y;
36   end transition;
37 end SystemSC;
38 end System;
```

2.2 Simulation experiment

The following simulation experiment illustrates the previous explained behaviour of the Pool-Billiard game. The parameter of the model are set in a manner, that all different types of events (1: collision of two balls, 2: reflection on a border, 3: disappearing in the hole) are present during the simulation experiment (see Fig. 7).

Figure 8 show the positions and the Figures 9 and 10 the reflection and collision events of the white and the black ball during a simulation period of 4 seconds.

After 0.2 seconds, the white ball collides with the black ball. After 1 second, the blackball is reflected twice in a short time period on the top side on the billiard-table and both balls collide again between its reflections. After 2.3 and 2.5 seconds the balls reflect on the left border. At 2.95 seconds the white ball

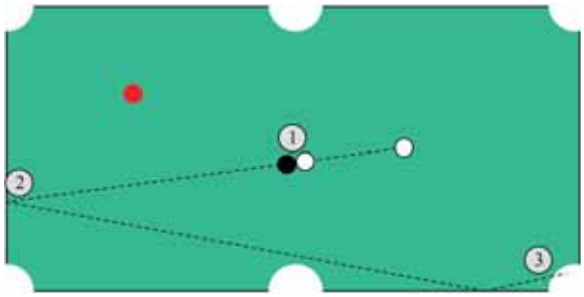


Figure 7. Event types in the Pool-Billard game

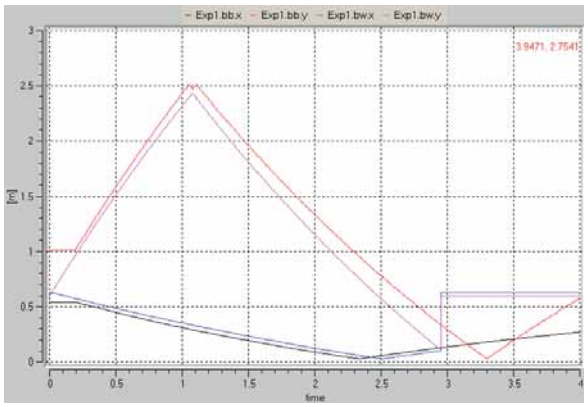


Figure 8. x- and y-positions of white and black ball.

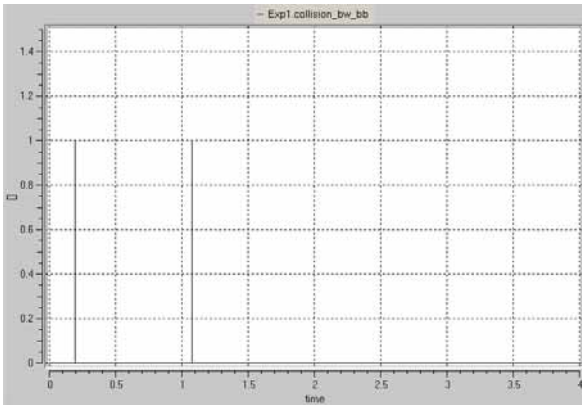


Figure 9. Collision events of white and black ball.

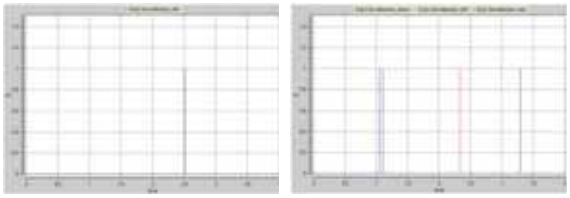


Figure 10. Reflection events of the white ball (left) and the black ball (right)

drops into the hole. At the end, the white ball is set again on its starting position.

3 Conclusions

The example of the modelling and simulation of a Pool-Billiard game has shown the advantages of the graphical modelling with UML^H for Modelica models. With UML^H, the design of a complex system model in Modelica begins with the drawing of its model structure. The class diagrams und the collaboration diagrams describe the object-oriented model construction and the statechart diagrams are used for the formulation of the event-driven model behaviour. If the Modelica tool supports code generation like MOSILAB, the Modelica code can be obtain automatically from the UML^H model. This pure code has to be filled up by the user with model equations (physical behaviour) of the modelled system.

References

[1] Nytsch-Geusen, C. et al.: MOSILAB: *Development of a Modelica based generic simulation tool supporting model structural dynamics*. Proceedings of the 4th International Modelica Conference, TU Hamburg-Harburg, Hamburg, 2005.

[2] Nytsch-Geusen, C. et al.: *Advanced modeling and simulation techniques in MOSILAB: A system development case study*. Proceedings of the 5th International Modelica Conference, Arsenal Research, Wien, 2006.

[3] MOSILAB-Webportal: `www.mosilab.de`.

[4] UML-Homepage: `www.uml.org`.

[5] Modelica-Homepage: `www.modelica.org`.

[6] Nordwig, A. et al.: MOSILA-Modellbeschreibungssprache, Version 2.0, Fraunhofer Gesellschaft, 2006.

Corresponding author: Christoph Nytsch-Geusen
Fraunhofer Institute for Computer Architecture and Software Technology
Kekuléstr. 7, 12489 Berlin, Germany
christoph.nytsch@first.fraunhofer.de

Accepted EOOLT 2007, June 2007
Received: August 10, 2007
Accepted: August 20, 2007

SNE 17/2, September 2007

Towards Unified System Modeling with the ModelicaML UML Profile

Adrian Pop, David Akhvlediani, Peter Fritzson
Programming Environments Lab, Sweden, *{adro, petfr}@ida.liu.se*

In order to support the development of complex products, modeling tools and processes need to support co-design of software and hardware in an integrated way. Modelica is the major object-oriented mathematical modeling language for component-oriented modeling of complex physical systems and UML is the dominant graphical modeling notation for software. In this paper we propose ModelicaML UML profile as an integration of Modelica and UML. The profile combines the major UML diagrams with Modelica graphic connection diagrams and is based on the System Modeling Language (SysML) profile.

Introduction

The development in system modeling has come to the point where complete modeling of systems is possible, e.g. the complete propulsion system, fuel system, hydraulic actuation system, etc., including embedded software can be modeled and simulated concurrently. This does not mean that all components are dealt with down to the very smallest details of their behavior. It does, however, mean that all functionality is modeled, at least qualitatively. In this paper, a UML profile for Modelica, named ModelicaML, is proposed. The ModelicaML UML profile is based on the OMG SysML™ (Systems Modeling Language) profile and reuses its artifacts required for system specification. SysML diagrams are also extended to support all Modelica constructs. We argue that with ModelicaML system engineers are able to specify entire systems, starting from requirements, continuing with behavior and finally perform system simulations.

1 SysML and Modelica

The Unified Modeling Language (UML) has been created to assist software development processes by providing means to capture software system structure and behavior. This evolved into the main standard for Model Driven Development [5]. The System Modeling Language (SysML) [4] is a graphical modeling language for systems engineering applications. SysML was developed by systems engineering experts, and was adopted by OMG in 2006. SysML is built on top of UML and tailored to the needs of system engineers by supporting specification, analysis, design, verification and validation of broad range of systems and system-of-systems.

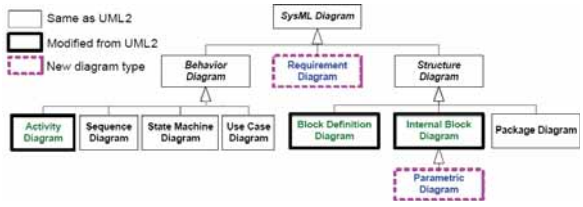


Figure 1. SysML diagram taxonomy.

The main goal behind SysML is to unify and replace different document-centric approaches in the system engineering field with a single systems modeling language. A single model-centric approach improves communication, assists to manage complex system design and allows its early validation and verification.

The taxonomy of SysML diagrams is presented in Fig. 1. For a full description of SysML see (SysML, 2006) [4]. The major SysML extensions compared to UML are:

- *Requirements diagrams* support requirements presentation in tabular or in graphical notation, allows composition of requirements and supports traceability, verification and "fulfillment of requirements".
- *Block diagrams* extend the Composite Structure diagram of UML2.0. The purpose of this diagram is to capture system components, their parts and connections between parts. Connections are handled by means of connecting ports which may contain data, material or energy flows.
- *Parametric diagrams* help perform engineering analysis such as performance analysis. Parametric diagrams contain constraint elements, which define mathematical equations, linked to properties of model elements.
- *Activity diagrams* show system behavior as data and control flows. Activity diagrams are similar to Ext. Functional Flow Block Diagrams, which are already widely used by system engineers. Activity decomposition is supported by SysML.

- *Allocations* are used to define mappings between model elements: For example, certain Activities may be allocated to Blocks (to be performed by the block).

SysML block definitions (Fig. 2) can include properties to specify block parts, values and references to other blocks. A separate compartment is dedicated for each of these features. To describe the behavior of a block the "Operations" compartment is reused from UML and it lists operations that describe certain behavior. SysML defines a special form of an optional compartment for constraint definitions owned by a block. A "Namespace" compartment may appear if nested block definitions exist for a block. A "Structure" compartment may appear to show internal parts and connections between parts within a block definition.

SysML defines two types of ports: standard ports and flow ports. Standard ports, which are reused from UML, are service-oriented ports required or provided by a block. Flow ports specify interaction points through which items may flow between blocks, and between blocks and environment. A flow port definition may include single item specification or complex flow specification through the FlowSpecification interface; flow ports define what "can" flow between the block and its environment. Flow direction can be specified for a flow port in SysML. SysML also defines a notion of Item flows that specify "what" does flow in a particular usage context.

1.1 Modelica

Modelica [2] [3] is a modern language for equation-based object-oriented mathematical modeling primarily of physical systems. Several tools, ranging from open-source as OpenModelica [1], to commercial like Dymola [11] or MathModelica [10] support the Modelica specification.

The language allows defining models in a declarative manner, modularly and hierarchically and combining

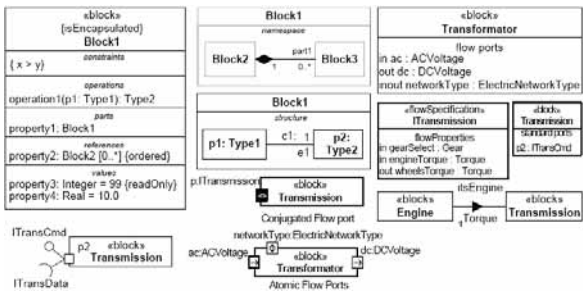


Figure 2. SysML block definitions

various formalisms expressible in the more general Modelica formalism. The multidomain capability of Modelica allows combining electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model. In short, Modelica has improvements in several important areas:

- *Object-oriented mathematical modeling*. This technique makes it possible to create model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.
- *Physical modeling of multiple application domains*. Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks with fixed input/output causality. In Modelica the structure of the model naturally correspond to the structure of the physical system in contrast to block-oriented modeling tools.
- *Acausal modeling*. Modeling is based on *equations* instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases reusability of model components, since components adapt to the data flow context in which they are used.

Hierarchical system architectures can easily be described with Modelica thanks to its powerful component model. The *Components* are connected via the *connection mechanism* realized by the Modelica system, which can be visualized in connection diagrams. The *component framework* realizes components and connections, and ensures that communication works over the connections. For systems composed of *acausal* components with behavior specified by equations, the direction of data flow, i.e., the *causality* is initially unspecified for connections between those components and the causality is automatically deduced by the compiler when needed. Components have well-defined *interfaces* consisting of ports, also known as *connectors*, to the external world. A component may internally consist of other connected components, i.e., *hierarchical* modeling as in Fig. 3.

1.2 SysML vs. Modelica

The System Modeling Language (SysML) has recently been proposed and defined as an extension of UML targeting at systems engineers. As previously stated, the goal of SysML is to unify different ap-

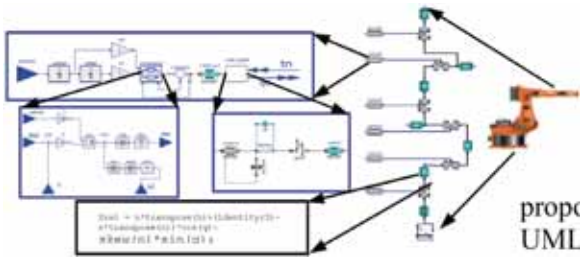


Figure 3. Hierarchical model of an industrial robot.

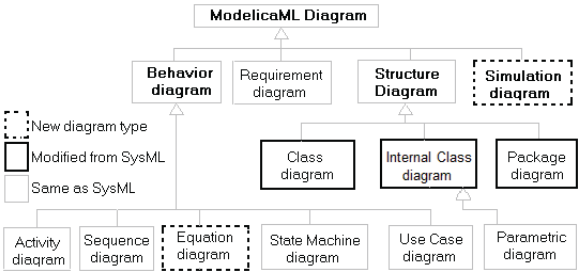


Figure 4. ModelicaML diagrams overview

proaches and languages used by system engineers into a single standard. SysML models may span different domains, for example, electrical, mechanical and software. Even if SysML provides means to describe system behavior like Activity and State Chart Diagrams, the precise behavior can not be described and simulated. In that respect, SysML is rather incomplete compared to Modelica.

Modelica also, was created to unify and extend object-oriented mathematical modeling languages. It has powerful means for describing precise component behavior and functionality in a declarative way. Modelica models can be graphically composed using Modelica connection diagrams which depict the structure of designed system. However, complex system design is more that just a component assembly. In order to build a complex system, system engineers have to gather requirements, specify system components, define system structure, define design alternatives, describe overall system behavior and perform its validation and verification.

2 ModelicaML: a UML profile for Modelica

ModelicaML reuses several diagrams types from SysML without any extension, extends some of them, and also provides several new ones. The ModelicaML diagram overview is shown in Fig. 4. Diagrams are grouped into four categories: Structure, Behavior, Simulation and Requirement. In the following we present the most important ModelicaML profile diagrams. The full description of the ModelicaML profile is presented in [8]. The most important properties of the ModelicaML profile are outlined in the following:

- The ModelicaML profile supports modeling with all Modelica constructs and properties i.e. restricted classes, equations, generics, discrete variables, etc.
- Using ModelicaML diagrams it is possible to describe multiple aspects of a system being designed and thus support system development process phases such as requirements analysis, design, implementation, verification, validation and integration.
- ModelicaML is partly based on SysML, but reuses and extends its elements.
- The profile supports mathematical modeling with equations since equations specify behavior of a (Modelica) system. Algorithm sections are also supported.
- Simulation diagrams are introduced to model and document simulation parameters and results in a consistent and usable way.
- The ModelicaML meta-model is consistent with SysML in order to provide SysML-to-ModelicaML conversion.

Three SysML diagram types have been partly reused and changed for the ModelicaML profile. The rest of the diagram types we used in ModelicaML unchanged:

- The SysML Block Definition Diagram has been updated and renamed to Modelica Class Diagram.
- The SysML Internal Block Diagram has been updated and renamed to Modelica Internal Class Diagram (some of the SysML constructs are disabled).
- The Package Diagram has been changed in order to fully support the Modelica language (i.e. Modelica package constants, Generic Packages, etc).
- Other SysML diagram types such as Use Case Diagram, Activity Diagrams and Allocations, and State Machine Diagrams are included in ModelicaML without modifications. ModelicaML re-

uses Sequence Diagrams from SysML and changes the semantics of message passing. Modelica doesn't support method declaration within a single class but supports declaration of functions as a restricted class type.

Thus, the following diagram types are available in the ModelicaML profile:

- The *Modelica Class Diagram* usually describes class definitions and their relationships such as inheritance and containment.
- The *Modelica Internal Class* Diagram describes the internal class structure and interconnections between parts.
- The *Package Diagram* groups logically connected user defined elements into packages. In ModelicaML the primarily purpose of this diagram is to support the specifics of the Modelica packages.
- Activity, Sequence, State Machine, Use Case, Parametric and Requirements diagrams have been reused without modification from SysML.
- Two new diagrams, *Simulation Diagram* and *Equation Diagram*, not present in SysML, have been included in the ModelicaML profile.

2.1 Package diagram

A UML Package is a general purpose model element for grouping other elements within a separate namespace. With a help of packages, designers are able group elements to correspond to different structures/views of a system. ModelicaML extends UML packages in order to support Modelica packaging features, in particular: package inheritance, generic packages, constant declaration within a package, package "instantiation" and renaming import (see [2] for Modelica packages details).

A diagram which contains package elements and their relationships is called a Package Diagram. Modelica packages have a hierarchical structure containing package elements as nodes. In Modelica, packages are used to structure model elements into libraries. A snapshot of the Modelica Standard Library hierarchy is shown in Fig. 5 using UML notation. Package nodes in the hierarchy are connected via the package containment link as in the example in Fig. 6.

2.2 Modelica class diagram

Modelica uses restricted classes such as `class`, `model`, `block`, `connector`, `function` and `record` to describe a system. Modelica classes have essentially the same semantics as SysML blocks specified in [4] and provide a general-purpose capability to model systems as hierarchies of modular components. ModelicaML extends SysML blocks by defining features which are relevant or unique to Modelica. The purpose of the Modelica Class Diagram is to show features of Modelica classes and relationships between classes. Additional kind of dependencies and associations between model elements may also be shown in a Modelica Class Diagram. For example, behavior description constructs – equations, may be associated with particular Modelica Classes. The detailed description of structural features of ModelicaML is provided below. ModelicaML structural extensions are defined based on the SysML block definition outlined in section 2.

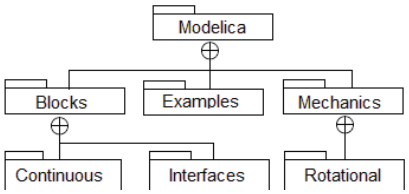


Figure 5. Package hierarchy modeling

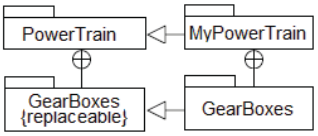


Figure 6. Package hierarchy modeling.

Modelica Class Definiton

The graphical notation of ModelicaML class definitions is shown in Fig. 7. Each class definition is adorned with a stereotype name that indicates the class type it represents. The ModelicaML Class Definition has several compartments to group its features: parameters, parts, variables. We designed the parame-

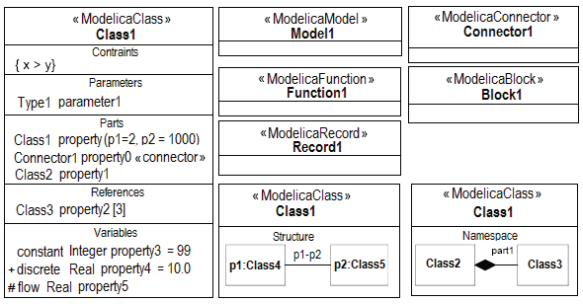


Figure 7. ModelicaML class definitions

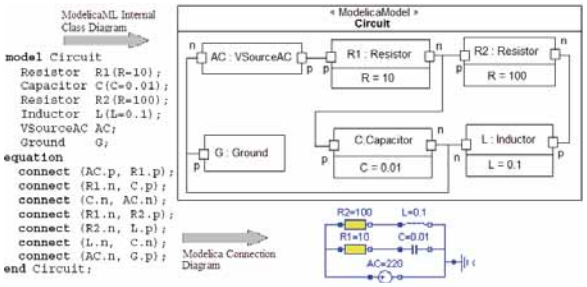


Figure 8. ModelicaML internal class vs. ModelicaML connection diagram

ters compartment separately from variables because the parameters need to be assigned values in order to simulate a model (see the Simulation Diagram later on). Some compartments are visible by default; some are optional and may be shown on ModelicaML Class Diagram with the help of a tool. Property signatures follow the Modelica textual syntax and not the SysML original syntax, reused from UML. A ModelicaML/SysML tool may allow users to choose between UML or Modelica style textual signature presentation. Using Modelica syntax on a diagram has the advantage of being more compatible with Modelica and being more straightforward for Modelica users. The Modelica syntax is quite simple to learn even for users not acquainted with Modelica.

ModelicaML provides extensions to SysML in order to support the full set of Modelica constructs and features. For example, ModelicaML defines unique class definition types ModelicaClass, ModelicaModel, ModelicaBlock, ModelicaConnector, ModelicaFunction and ModelicaRecord that correspond to class, model, block, connector, function and record restricted Modelica classes. We included the Modelica specific restricted classes because a modeling tool needs to impose their semantic restrictions (for example a record cannot have equations, etc).

Modelica internal class diagram

The Modelica Internal Class Diagram is based on the SysML Internal Block Diagram but the connections are based on ModelicaConnector. The Modelica Class Diagram defines Modelica classes and relationships between classes, like generalizations, association and dependencies, whereas a Modelica Internal Class Diagram shows the internal structure of a class in terms of parts and connections. The Modelica Internal Class Diagram is similar to Modelica connection diagram, which presents parts in a graphical (icon) form. An example Modelica model presented as a Modelica Internal Class diagram is shown in Fig. 8.

Usually Modelica models are presented graphically via Modelica connection diagrams (Fig. 8, bottom). Such diagrams are created by the modeler using a graphic connection editor by connecting together components from available libraries. Since both diagram types are used to compose models and serve the same purpose, we briefly compare the Modelica connection diagram to the Modelica Internal Class Diagram. The main advantage of the Modelica connection diagram over the Internal Class Diagram is that it has better visual comprehension as components are shown via domain-specific icons known to application modelers. Another advantage is that Modelica library developers are able to predefine connector locations on an icon, which are related to the semantics of the component. In the case of a ModelicaML Internal Class Diagram a SysML/ModelicaML tool should somehow point out at which side of a rectangular presentation of a part to place a port (connector).

One of the advantages of the Internal Class Diagram is that it directly supports nested structures. However, nested structures are also available behind the icons in a Modelica connection diagram, thus using the drawing area more effectively.

The main advantage of the Internal Class Diagram is that it highlights top-level Modelica model parameters and variables specification in separate compartments.

Other SysML elements, such as Activities and Requirements which do not exist in Modelica but are very important for additional model specification can be combined with both Internal Class Diagram and Modelica connection diagrams.

2.3 Parametric diagrams vs. equation diagrams

SysML defines Constraint blocks which specify mathematical expressions, like equations, to constrain physical properties of a system. Constraint blocks are defined in the Block Definition diagram and can be packaged into domain-specific libraries for later reuse. There is a special diagram type called Parametric Diagram which relates block parameters with certain constraints blocks. The Parametric Diagram is included in ModelicaML without any modifications to keep the compatibility with SysML.

The Modelica class behavior is usually described by equations, which also constrain Modelica class pa-

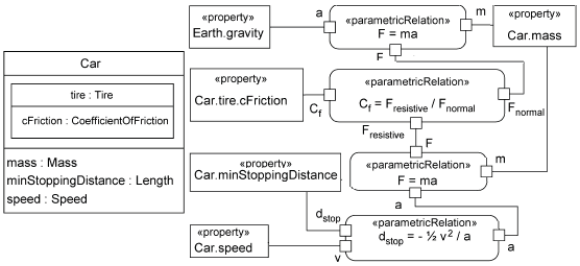


Figure 9. Parametric diagram example

rameters, and have a domain-specific usage. SysML constraint blocks are less powerful means of domain model description than Modelica equations. Modelica equations include some type of equations, which cannot be modeled using Constraint blocks, i.e.: `if`, `for`, `when` equations. Also, modeling complexity is an issue, as for example in Fig. 9 there are only four equations, and the diagram is already quite complex. However, grouping constraint blocks into libraries can be useful for system engineers who use Modelica and SysML. SysML Parametric diagram may be used during the initial design phase, when equations related to a class are being identified using Parametric Diagrams and finally associated (via an Equation Diagram) with a Modelica class or set of classes.

In Fig. 10, Fig. 11 we present examples of behavior specification via Equation Diagrams in ModelicaML. Equations do not prescribe a certain data flow direction which means that the order in which equations appear in a model do not influence their meaning and semantics. The only requirement for a system of equations is to be solvable. For further details about Modelica equations, see [2]. Besides simple equality equations, Modelica allows other kind of equations be presented within a model. For each of such kind of equations (i.e. when/if/initial equations) ModelicaML

```
partial class TwoPin
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;

class Resistor
  extends TwoPin;
  parameter Real R
       (unit = "Ohm");
equation
  R * I = v;
end Resistor;
```

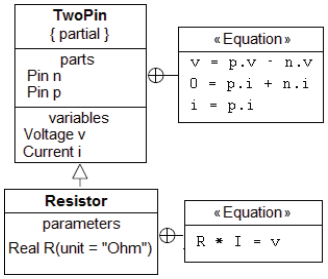


Figure 10. Equation modeling example with a Modelica Class Diagram.

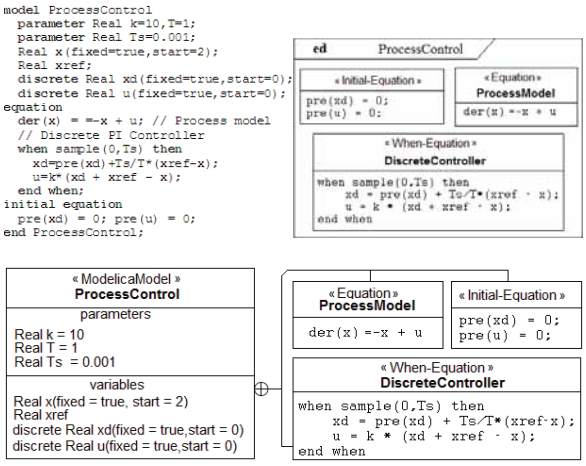


Figure 11. ModelicaML nested/extern Equation diagrams

defines a graphical construct. It's up to designer to decide whether to use simple equations block representation or specific construct for equation modeling. Algorithm sections are modeled similar to equations, as text.

With a help of Equation Diagram top-down modeling approach is applied to behavior modeling. First, the primarily equations may be captured, then conditional constructs applied, equations text description substituted with mathematical expressions or even equations refactored by moving to other classes. In the similar way as Modelica classes are grouped by physical domain libraries, common equations can be packaged into domain-specific libraries and be reused during a design process. Moreover, equation constructs shown on Equation Diagram can be linked to Activity elements or with Requirement elements to show that a specific requirement has been fulfilled.

2.4 Simulation diagram

ModelicaML introduces a new diagram type, called Simulation Diagram (Fig. 12), used for simulation modeling. Simulation is usually performed by a simulation tool which allows parameter setting, variable selection for output and plotting. The Simulation Diagram may be used to store any simulation experiment, thus helping to keep the history of simulations and its results. When integrated with a modeling and simulation environment, a simulation diagram may be automatically generated.

The Simulation Diagram provides facilities for simulation planning, structured presentation of parameter passing and simulation results. Simulations can be run directly from the Simulation Diagram. Association of simulation results with requirements from a

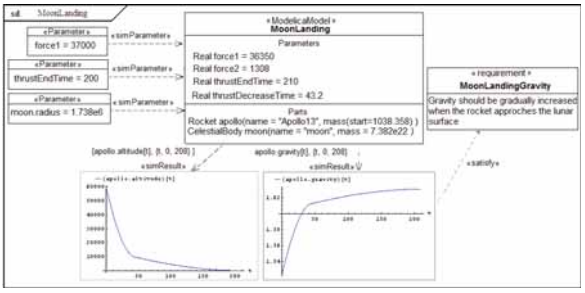


Figure 12. Simulation diagram example

domain expert and additional documentation (e.g. by: Note, Problem Rationale text boxes of SysML) are also supported by the Simulation Diagram. The Simulation Diagram introduces new diagram elements: "Parameter" element and two stereotyped dependency associations, "simParameter" and "simResults". Parameter values are associated with a class via simParameter for a simulation. Simulation results are associated with a model via simResults which specify which variable is to be plotted and for what time interval.

For simulation purposes, the Simulation Diagram can be integrated with any Modelica modeling and simulation environment. We are currently in the process of designing a ModelicaML development environment which integrates with the OpenModelica modeling and simulation environment.

3 Conclusion and future work

In this paper we propose the ModelicaML profile that integrates Modelica and UML. UML Statecharts and Modelica have been previously integrated, see e.g. [9][15]. SysML is rather new but it was already adopted for system on chip design [13] evaluated for code generation [14] or extended with bond graphs support [12].

The support for Modelica in ModelicaML allows precisely defining, specifying and simulating physical systems. Modelica provides the means for defining behavior for SysML block diagrams while the additional modeling capabilities of SysML provides additional modeling and specification power to Modelica (e.g. requirements and inheritance diagrams, etc).

As a future project we plan to implement an Eclipse-based [6] graphical editor for ModelicaML as a part of our Modelica Development Tooling (MDT) [7].

References

[1] P Fritzson, P Aronsson, H Lundval, K Nyström, A Pop, L Saldamli, and D Broman. *The OpenModelica Modeling, Simulation, and Software Development Environment*. In Simulation News Europe, 44/45, Dec 2005. `http://ww.ida.liu.se/projects/OpenModelica`.

[2] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pp., Wiley-IEEE Press, 2004. See also: `http://www.mathcore.com/drmodelica/`

[3] The Modelica Association. *The Modelica Language Specification Version 2.2.*

[4] OMG, *System Modeling Language, (SysML),* www: `http://www.omgsysml.org`

[5] OMG : Guide to Model Driven Architecture: The MDA Guide v1.0.1

Corresponding author: Adrian Pop,
Programming Environments Lab,
Department of Computer and Information Science,
Linköping Univ. SE-581 83 Linköping, Sweden
adrpo@ida.liu.se

Accepted EOOLT 2007, June 2007
Received: August 10, 2007
Accepted: August 20, 2007

Hybrid Dynamics in Modelica: Should all Events be Considered Synchronous

Ramine Nikoukhah, INRIA-Rocquencourt, France, *ramine.nikoukhah@inria.fr*

The Modelica specification is ambiguous as to whether all the events are synchronous are not. Different interpretations are possible leading to considerable differences in the ways models should be constructed and compilers developed. In this paper we examine this issue and show that there exists an interpretation that is more appropriate than others leading to more efficient compilers. It turns out that this interpretation is not the one currently adopted by Dymola but it is closely related to the Scicos formalism.

SNE 17/2, September 2007

Introduction

Modelica (`www.modelica.org`) is a language for modeling physical systems. It has been originally developed for modeling systems obtained from the interconnection of components from different disciplines such as electrical circuits, hydraulic and thermodynamics systems, etc. These components are represented symbolically in the language providing the compiler the ability to perform symbolic manipulations on the resulting system of differential equations. This allows the usage of acausal components (equation based) without loss of performance.

But Modelica is not limited to continuous-time models [1]; it can be used to construct hybrid systems, i.e., systems in which continuous-time and discrete-time components interact. Modelica specification [2] tries to define the way these interactions should be interpreted and does so by inspiring from the formalism of synchronous languages. Synchronous languages however deal with events, i.e., discrete-time dynamics. So in the context of Modelica, the concept of synchronism had to be extended to encompass continuous-time dynamics as well. It is exactly this extension which is the subject of this paper.

Scicos (`www.scicos.org`) is a modeling and simulation environment for hybrid systems. It is free software, included in the scientific software package Scilab (`www.scilab.org`). Scicos formalism is based on the extension of synchronous languages, in particular Signal [3], to the hybrid environment. The class of models that Scicos is designed for is almost the same as that of Modelica. So it is not a surprise that Modelica and Scicos have many similar features and confront similar problems. Modelica has many advantages for modeling continuous-time dynamics, especially thanks to its ability to represent models in symbolic form, whereas the Scicos formalism has been specifically designed to allow high performance code generation of discrete-time dynamics.

In this paper, we examine the specification of hybrid dynamics in Modelica and propose an interpretation that is fully compatible with the Scicos formalism. This interpretation, which is not contradictory with the official specification, allows us to obtain an efficient compiler/code generator for Modelica inspired by the Scicos compiler.

Here we start with a flat model (obtained from the application of a front-end compiler), and consider only the problems concerning the design of the phase one of a back-end compiler. This phase breaks down the code into independent asynchronous parts each of which can be compiled separately in phase two. Phase two will be presented in a subsequent paper.

1 Conditioning and sub-sampling in Modelica

If a model contains no conditioning and all of its parts function at the same rate, then back-end compilation would be a simple task. But in most real life applications, models contain different dynamics resulting from the inter-connection of heterogeneous systems. A model of such a system would often include conditioning and sub-sampling. We use the term conditioning for a change in the model conditioned on the value of a variable (for example `if a>0 then`) and the term sub-sampling for the construction of a new, not necessarily regular, clock from a faster clock.

The `when-elsewhen` and `if-then-else` clauses are the basic language constructs in Modelica for performing conditioning and sub-sampling. The description of the ways these constructs function is ambiguous in the Modelica specification. Comparing with the Scicos formalism, we can consider that Mode-

lica's `if-then-else` clause does conditioning and when does sub-sampling. But the situation is somewhat more complex because `when` plays two different roles. And, we need to distinguish these two different types of `when` clauses. But before, we need to examine the notion of synchronism in Modelica.

1.1 Synchonous versus simultaneous

In our interpretation of the Modelica specification, two events are considered synchronous only if they can be traced back to a single event source. For example, in the following model:

```
when sample(0,1) then
   d = pre(d)+1;
end when;
when d>3 then
   a = pre(a)+1;
end when;
```

the event `d>3` is synchronous with the event `sample(0,1)`. The former is the source of the latter. But in

```
der(x) = x;
when sample(0,1) then
   d = pre(d)+1;
end when;
when x>3 then
   a = pre(a)+1;
end when;
```

the two events are not synchronous. There is no unique source of activation at the origin of these events. So these events are considered asynchronous even if the two events are activated simultaneously; even if we can prove mathematically that they always occur simultaneously.

Our basic assumption is that events detected by the zero-crossing mechanism of the numerical solver (or an equivalent mechanism used to improve performance) are always asynchronous. So even if they are detected simultaneously by the solver, by default they are treated sequentially in an arbitrary order. In particular, in the model:

```
when sample(0,1) then
  b = a;
end when;
when sample(0,1) then
  a = b+1;
end when;
```

the variables a and b can be evaluated in any order.

Dymola on the other hand assumes that all events are synchronous. In particular it assumes that all the equations in both when clauses may have to be satisfied simultaneously. That is why Dymola finds an algebraic loop in this example.

To see the way Dymola proceeds, consider the following example:

```
equation
   der(x) = 1;
   der(y) = 1;
   when (x>2) then
      z = pre(z)+3;
      v = u+1;
   end when;
   when (y>2) then
      u=z+1;
   end when;
```

The simulation shows that the equations (assignments) are ordered as follows:

```
z=pre(z)+3; u=z+1; v=u+1;
```

this means that the content of a `when` clause is split into separate conditional clauses. In stark contrast, in our interpretation of the Modelica specification, the code within an asynchronous `when` clause is treated synchronously and never broken up. Both interpretations are valid and consistent; however our interpretation has many advantages as we will try to show here.

At first glance, the non determinism that may be encountered in our approach when two zero-crossing events occur simultaneously may seem unacceptable. However, treating two simultaneous zero-crossings as synchronous is not a solution because it is not robust. Indeed, when dealing with nonlinear and complex models, there is no guarantee that the numerical solver would detect two zero-crossings simultaneously even if theoretically they are simultaneous. In general one is detected slightly before or after the other. And in any case, in most cases treating such an accidental synchronism is not of any use for the construction of the model. Even if the model depends for some reason on the simultaneous detection of two events by the solver, a mechanism should be provided by the language to specify explicitly what should be done in that case. One way would be to introduce a switchwhen clause [4], which can be used to explicitly specify what equations are activated in every possible case. The possible cases when we have, for example, two zero-crossings are: the first surface has crossed but not the second, the second has crossed but not the first and finally both surfaces have crossed zero together.

Dymola's interpretation imposes constraints, which in most cases are useless. Moreover, when all zero-

crossing events are considered synchronous, the complexity of static scheduling increases with the number of zero-crossings. The solution based on using the switchwhen clause allows the user to specify explicitly what possible synchronisms must be considered. It turns out that in most cases, no synchronism is to be considered.

1.2 Primary and secondary when clauses

So far we have seen two types of when clauses, or more specifically when clauses based on two types of events: events depending on variables evolving continuously in time such as time>3 or x<2 where x is a continuous variable; and events depending on discrete variables. when clauses conditioned on events of the former type are called *primary*, the latter ones are called *secondary*.

An event associated with a secondary when clause is necessarily synchronous with events associated to one or more primary when clauses. These primary clauses are those in which the discrete variables involved in the definition of the event are defined.

But not all when clauses can easily be classified as primary or secondary. Let us consider a simple example:

```
when sample(0,1) then
   d = pre(d)+j;
   c = b;
end when;
when time > d then
   b = a;
end when;
```

The question is whether or not the above two when clauses are primary or not. Clearly the first one is, but the second hides in reality two distinct when clauses that is because the event time>d can be activated in two different ways:

- time increases and crosses d continuously (zero-crossing event so asynchronous),
- at a sample time d jumps, activating the time>d condition; this event is clearly synchronized with sample(0,1).

We call such when clauses *mixed*. We handle this situation by implementing the simulation in such a way that time>d is activated only when time crosses continuously d and placing a duplicate of the content of this when where d is defined within a condition that guarantees that the content is activated only if time>d is activated due to a jump:

```
when sample(0,1) then
   d = pre(d)+j;
   c = b;
   if ((time>d) and not(time>pre(d))) then
      b = a;
   end if;
end when;
when time>d then
   b = a;
end when;
```

The second solution amounts to considering that a clause such as when c>0 where c is a continuous variable is activated only if c crosses zero continuously (the way that is detected by zero-crossing mechanisms built into numerical solvers such as LSODAR or DASKR). This seems to be an appropriate way to handle mixed when clauses, however to stay compatible with the Modelica specification, at a pre-compilation phase, the content of these clauses must be duplicated as explained above.

Note that the code we obtain after the pre-compilation phase is not correct according to the Modelictcification (because b is defined twice). This however is not a problem because this code is only used within the compiler. But in any case, we consider this restriction too restrictive and we think it should be relaxed. This will be discussed later.

There still remains a situation that needs clarification. Consider the following example:

```
discrete Real a(start=0);
Real x(start=0);
equation
   der(x)=0;
   when x>3 then
      a = pre(a)+1;
   end when;
   when time>2 then
      reinit(x,x+4);
   end when;
```

Here x is a continuous variable, but it is also discrete because at time 2 it jumps from 0 to 4 (activation of reinit). This jump activates the content of the first when. The reinit primitive in this case must be considered as a definition of "discrete" x, so following the rule discussed previously, the content of the clause when x>3 must be copied inside the other when:

```
discrete Real a(start=0);
Real x(start=0);
equation
   der(x) = 0;
   when x>3 then
```

```
      a = pre(a)+1;
   end when;
   when time>2 then
      reinit(x,x+4);
      if edge(x>3) then
         a = pre(a)+1;
      end if;
   end when;
```

This transformation is just a special case of the situation we have considered previously. To see this more clearly, note that

```
when time>2 then
   reinit(x,x+4);
end when;
```

should really be expressed as follows

```
when time>2 then
   x = pre(x)+4;
end when;
```

1.3 Restrictions on the use of when and if

Modelica imposes hard constraints on the usage of when and if-then-else clauses.

In the case of when, a variable is not allowed to be defined in two when clauses. For example, the following code is not allowed in an equation section:

```
when sample(0,1) then
   b = pre(b)+1 ;
end when;
when time>3.5 then
   b = 0;
end when;
```

According to the specification, this can lead to a contradiction if the two when clauses are activated at the same time. This statement would make sense if the two when clauses were synchronous but not in this case. Lifting this restriction, in the case of primary when clauses, is without danger and facilitates the task of modeling in many situations. However, it creates an important difference as far some interpretation of the primitive pre is concerned. With the current restriction, we are sure that in the following code:

```
when sample(0,1) then
   b = pre(b)+1 ;
end when;
```

pre(b) is the previous value of b defined by b=pre(b)+1 the last time this when clause was activated, i.e. one unit of time before. So without even having to examine the rest of the code, we can be sure that b indicates the time. This will no longer be true if the constraint is lifted; consider:

```
when sample(0,1) then
   b = pre(b)+1;
end when;
when sample(.5,1) then
   b = pre(b)+1;
end when;
```

In this case the value of b used to update it in each clause is computed by the instruction in the other clause. But this is not a problem as long as the rules are clear.

We thus propose the following modifications: this restriction be lifted for primary when clauses and this restriction be lifted in all when clauses as long as the equations defining common variables are identical (such identical equations can arise in transformations applied by the compiler which includes duplicating parts of the code). For example for all conditions c1, c2 (synchronous or not), accept:

```
equation
   when c1 then
      b = a;
   end when;
   when c2 then
      b = a;
   end when;
```

The second modification may seem strange. Indeed why would a model contain identical statements in synchronous when clauses. The reason is that our Modelica compiler performs a series of transformations each one generating a new Modelica code from a Modelica code in which such a situation may come up (this happens in particular when processing the union of events construct, see Section 1.6). By lifting this restriction, we make sure that we obtain a valid Modelica code at every stage. But a specific test must be applied to the original model to issue at least a warning to the user for such cases.

Another important restriction concerns the use of elsewhen. The Modelica specification states that all the branches of a when-elsewhen clause must define the same set of variables. We don't believe this constraint is justified. This constraint is probably a consequence of a similar condition on the use of if-then-else clauses. Indeed Modelica imposes that the number of equations in different branches of such a clause be identical. This may be acceptable as far as continuous-time variables are concerned—removing the restriction in the continuous case makes it possible to model Simulink's enabled Super Blocks in Modelica—, but it is not for discrete variables. So we propose to lift this restriction and accept models including for example the following code:

```
equation
   when sample(0,1) then
      if u>0 then
         v = 1;
      end if;
   end when;
```

Normally in Modelica we should have an `else` branch defining `v`. Note that our proposal is not just an editing facility (i.e., a way to avoid writing code which can be added in automatically later); this code is not equivalent to

```
equation
   when sample(0,1) then
      if u>0 then
         v = 1;
      else
         v = pre(v);
      end if;
   end when;
```

In the absence of the `else` branch, the variable `v` is sub-sampled. This would not be the case if `v=pre(v)` were used. Even if the simulation result would be the same, the construction by sub-sampling leads to the generation of more efficient code. Lifting this restriction is again important for transformed models. A specific test can be used on the original model to impose the constraint if desired.

1.4 Continuous time dynamics

Our objective is to reduce the Modelica code into a number of asynchronous `when` clauses each of which can be treated separately. The continuous dynamics is no exception. What we call continuous dynamics includes everything within the `equation` section but outside `when` clauses. These equations are always active (Scicos terminology) even when a `when` clause is activated. So these equations are synchronous with all the `when` clauses.

The way this situation is handled in Scicos is to introduce a fictitious clock generating a continuous activation signal. To do the same in Modelica amounts to defining a special `when` clause:

```
when continuous then
```

the content of which would be active all the time except at event instances associated to other `when` clauses. Doing so allows us to consider the `continuous` event as asynchronous with the rest and treat this `when` clause as primary. To preserve the dynamics of the original model, the continuous dynamic equations must also be copied inside all the when clauses. For example:

```
equation
   y = sin(time);
   der(x) = y;
   when x<.2 then
      a=y;
   end when;
```

becomes

```
equation
   when continuous then
      y = sin(time);
      der(x) = y;
   end when;
   when x<.2 then
      y = sin(time);
      der(x) = y;
      a=y;
   end when;
```

During the simulation, the content of the `when continuous` clause is used to respond to the queries of the numerical solver, and in particular to generate the value of `der(x)` in this case. In other `when` clauses, the equations defining derivative values can be dropped, especially in the explicit case. In the implicit case (DAE case), the computation of the derivatives can be used to help the re-initialization of the solver.

The point to retain from this section is that the clause `when continuous` is primary and that its content can be treated like any other.

1.5 Initial conditions

In Modelica, variables can be initialized in different ways but in a flat model (after the application of the front end), they should all be grouped within a single when clause:

```
when initial then
   a = 0;
   d = 3;
   ...
end when
```

This would be a primary `when` clause and would contain the initialization of all discrete and continuous variables.

A `when terminal` clause can similarly be used to specify whatever needs to be done at the end of the simulation.

1.6 Union of events

The `when` and `elsewhen` clauses can be activated at the union of events. In Modelica the syntax is as follows:

```
when {c1, c2, c3} then
   < eq1 >
   < eq2 >
end when;
```

In this case, c1, c2, c3 may be synchronous or not. Note that the content of synchronous when clauses should not be executed more than once. For example in:

```
when sample(0,1) then
   d = pre(d)+1;
end when;
when {d>2, 2*d>4} then
   a = pre(a)+1 ;
end when;
```

a must be incremented only once, passing from zero to one. But in:

```
when sample(0,1) then
   d = pre(d)+1;
end when;
when sample(0,1) then
   e = pre(e)+1;
end when;
when {d>2, e>2} then
   a = pre(a)+1;
end when;
```

a is incremented twice (its value must jump from zero to two). But Dymola considers the d>2 and e>2 synchronous and increments a just once in this case. Similarly in:

```
when sample(0,3) then
   d = pre(d)+1;
end when;
when time>=3 then
   e = pre(e)+1;
end when;
when {d>1, e>0} then
   a = pre(a)+1;
end when;
```

in Dymola d>1, e>0 are synchronous (a is incremented only once at time 3). As we have said previously, we think that this interpretation must be avoided.

The counterpart of the union of events is the sum of activation signals in Scicos. The two formalisms coincide perfectly in this case.

In one of the early phases of the compilation of Modelica code, we propose the following transformation which removes all event unions. For example the first when clause presented in this section would be transformed as follows:

```
when c1 then
   < eq1 >
   < eq2 >
end when;
when c2 then
   < eq1 >
   < eq2 >
end when;
when c3 then
   < eq1 >
   < eq2 >
end when;
```

This code is correct if we take into account all the modifications suggested previously whether the ci, i=1,2,3, are synchronous or not.

2 Back-end compiler

The back-end compiler can be divided into two phases. The objective of the first phase is to transform the model into one in which all the when clauses are primary. This will allow us to generate, in phase two, static code for each one independently of the others.

Consider the following example:

```
when time>3 then
   d = pre(d)+1;
end when;
when d>3 then
   a = pre(a)+1;
end when;
when a>3 then
   b = a;
end when;
```

We want to remove the secondary when clauses. Clearly in this case we have to remove the last two when clauses. We pick one (say when a>3) and copy its content everywhere the variables involved in the definition of the corresponding event are computed. In this case the only variable involved is a, which is defined in the second when clause:

```
when time>3 then
   d = pre(d)+1;
end when;
when d>3 then
   a = pre(a)+1;
   if edge(a>3) then
      b > a;
   end if;
end when;
```

and then

```
when time>3 then
   d = pre(d)+1;
   if edge(d>3) then
      a = pre(a)+1;
      if edge(a>3) then
         b > a;
      end if;
   end if;
end when;
```

This example shows how secondary when clauses can be removed to obtain a single primary when clause at the end. If the model contains more than one primary when clause, the procedure would still be the same as illustrated in the following example:

```
when time>2 then a = 1; end when;
when time>3 then b = pre(b)+1; end when;
when a>b then c = 1; end when;
```

In this case the first two when clauses are primary. We now remove the secondary when:

```
when time>2 then
   a = 1;
   if edge(a>b) then
      c = 1;
   end if;
end when;
when time>3 then
   b = pre(b)+1;
   if edge(a>b) then
      c = 1;
   end if;
end when;
```

In this example, a variable is defined twice in two different primary (so asynchronous) when clauses. Clearly, this is not a problem. But the application of the transformation, can also lead to a variable being defined more than once in the same when clause. Let us examine the following example:

```
when time>2 then
   a = pre(a)+1;
end when;
when a>d then
   b = pre(b)+1;
end when;
when {a>2,b>2} then
   n = pre(n)+1;
end when;
```

We start by removing the operator "union of events". Then we remove the secondary clauses as previously described. We obtain (in two steps):

```
when time>2 then
   a = pre(a)+1;
   if edge(a>d) then
      n = pre(n)+1;
   end if;
   if edge(a>2) then
      b = pre(b)+1;
      if edge(b>2) then
         n = pre(n)+1;
      end if;
   end if;
end when;
```

This code, once edge replaced with its definition, may seem to be ordered properly and usable as a sequential code. But this is not the case since n=pre(n)+1, in some cases, can be executed twice instead of once. As discussed in the previous section, it is allowed to have a variable defined twice synchronously as long as the equations defining it are identical. This is of course the case here (this is the case in general when it happens because of the application of our transformations).The second phase of the compilation will transform the code into a sequential code correctly.

3 Conclusion

We have examined the notion of synchronism in Modelica and have shown that by abandoning the fully synchronous assumption, it is possible to design more efficient compilers without loss of rigor in the language specification. We have done that by proposing a methodology to implement the first phase of a back-end compiler. The second phase, which is closely related to the second phase of the Scicos compiler, will be presented in a future.

References

[1] M. Otter, H. Elmqvist, S. E. Mattsson. Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle. CACSD'99, Aug; 1999, Hawaii, USA.

[2] Modelica Association. Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification, version 2.2. 2005, available from www.modelica.org.

[3] Benveniste, P. Le Guernic, C. Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. Science of Computer Programming, 16, 1991, p. 103-149.

[4] R. Nikoukhah. Extensions to Modelica for efficient code generation and separate compilation, in Proc. EOOLT Workshop at ECOOP'07, Berlin, 2007.

[5] P. Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley-IEEE Press, 2003.

[6] S. L. Campbell, Jean-Philippe Chancelier, Ramine Nikoukhah. Modeling and Simulation in Scilab/Scicos, Springer, 2005.

Corresponding author: Ramine Nikoukhah
INRIA-Rocquencourt
BP 105, 78153 Le Chesnay Cedex, France
ramine.nikoukhah@inria.fr

Accepted EOOLT 2007, June 2007
Received: August 15, 2007
Revised: September 5, 2007
Accepted: September 20, 2007

TN

Enhancing Modelica towards Variable Structure Systems

Dirk Zimmer, ETH Zürich, Switzerland, *dzimmer@inf.ethz.ch*

This paper explains the motivation behind variable structure systems and analyses the current Modelica language with respect to those concerns. The major flaws and shortcomings are discussed to raise the awareness for the most relevant problem sets. Furthermore we sketch our current research activity in broad terms and explain our approach that consists of a new modeling language. Finally, a small example is presented.

1 Motivation

Many contemporary models contain structural changes at simulation run time. These systems are typically denoted by the collective term: variable structure systems. The motivations that lead to the generation of such systems are manifold:

- The structural change is caused by ideal switching processes. Classic examples are ideal switching processes in electric circuits, rigid mechanical elements that can break apart, e.g. a breaking pendulum or reconfiguration of robot models [4].
- The model features a variable number of variables: This issue typically concerns social or traffic simulations that feature a variable number of agents or entities, respectively.
- The variability in structure is to be used for reasons of efficiency: A bent beam should be modeled in more detail at the point of the buckling and more sparsely in the remaining regions.
- The variability in structure results from user interaction: When the user is allowed to create or connect certain components at run time, this usually reflects a structural change.

The term variable structure system turns out to be a rather general term that applies to a number of different modeling paradigms, such as adaptive meshes in finite elements, discrete communication models of flexible computer networks, etc. We focus on the paradigm that is represented by Modelica: declarative models that are based on DAEs with hybrid extensions. Within such a paradigm, a structural change is typically reflected by a change in the set of variables, and by a change in the set of relations (i.e., equations) between these time-dependent variables. These changes may lead to severe changes in the model structure. This concerns the causalization of the equation system, as well as the perturbation index of the DAE system.

A general modeling language supporting variable structure systems offers a number of important benefits. Such a potential language incorporates a general modeling methodology that enables the convenient capture of knowledge concerning variable structure systems, and provides means for organizing and sharing that knowledge both by industry and science. A corresponding simulator is a valuable tool for engineering and science education.

In concrete terms, our research is intended to aid the further extension of the Modelica framework. This benefits primarily the prevalent application areas of mechanics and electronics.

- Ideal switching processes in electronic circuits (resulting from ideal switches, diodes, and thyristors) can be more generally modeled. Occurring structural singularities can be handled at run time.
- The modeling of ideal transitions in mechanical models, like breaking processes or the transition from friction to stiction, become a more amenable task.

Additional applications may occur in domains that are currently foreign to Modelica. This might concern for instance: hybrid economic or social simulations that contain a variable number of entities or agents, respectively, and traffic simulations.

Finally, more elaborate modeling techniques become feasible. For instance multilevel models can be developed, whereby the appropriate level of detail is chosen at simulation run time in response to computational demands and/or level of interest.

2 Analysis of Modelica

Unfortunately, the modeling of variable structure systems within the current Modelica framework is very limited. This is partly due to a number of technical restrictions that mostly originate from the static treatment of the DAEs. Specific techniques, like inline-integrations [2] can help in certain situations, but they do not provide a general solution. Although the technical restrictions represent a major limiting factor, other issues need to be concerned as well. An

important problem is the lack of expressiveness in the Modelica-language.

To get a better understanding, we analyze the Modelica language with respect to the modeling of structural changes and list the most problematic points in the following subsections.

2.1 Lack of conditional declarations

Modelica is a declarative language that is based upon the declaration of equations, basic variables and sub-models. Modelica offers conditional blocks (i. e.: if, when) that enable the convenient formulation of changes in the system-equations. However, the declaration of variables or sub-models is kept away from these conditional blocks and is restricted to the unconditional header-section. Hence there is no mechanism for instance creation or removal at run-time (in fact, there exists a small mechanism for conditional declaration in Modelica that is supported by Dymola, but the conditions are based upon parameters and the way it is done restricts the access on such a conditional object to connect-statements).

2.2 No dynamic linking

The linking of an identifier to its instance is always static in Modelica. To conveniently handle objects that are created at run time, a dynamic linking of identifiers to their instances becomes desirable. Consequently, the linking must be assigned by the use of appropriate operators. Sub-models have now to be treatable as an entity.

2.3 Nontransparent type system

Such assignments that operate on complete model-instances also increase the emphasis on the type analysis like type-compatibility. Modelica is based on a structural type system [1] that represents a powerful and yet simple approach. Sadly, the actual type is not made evident in the language for a human reader since type members and non-type members mix in the header-section. Also the header section itself might be partitioned in different parts. Hence it becomes hard to identify the type of sub-models just by reading its declaration. This becomes a crucial issue when objects need to be treated dynamically.

2.4 Accessing the environment

Each model in Modelica is defined as a closed entity that cannot access by itself any outside variables. Whereas such a restriction is meaningful in most of the cases, it is inappropriate for certain tasks. One of these tasks is for instance the automatic connection of mass-holding objects to a gravity field. Modelica offers the concept of outer-models for this purpose. Unfortunately this approach is quite limited and represents not a feasible approach for more complex data-structures. At most, outer models could be used to create pools for mutual gravitational attraction [8] or potential collisions [3]. But to enable such pools, the single-pool members had to be manually assigned to an appropriate integer-ID. This is not a convenient solution.

The dynamic creation of sub-models increases the importance of a feasible solution for this task. When objects are created dynamically, they also need to be connected to other objects in their environment. Connections to other sub-models need to be established automatically at simulation time.

2.5 Insufficient handling of discrete events

Processes for the creation, removal and handling of dynamic instances represent discrete processes. Hence a powerful support for discrete-event handling is necessary. Modelica offers hybrid extension for such modeling tasks that are inspired by the synchronous data-flow principle [5]. However, for larger systems the current implementation may lead to an computational overkill and hence more elaborated concepts are needed.

The creation and connection have to be managed by discrete events. During such a construction process, singular equation systems may temporarily occur. However, they are not meant to be evaluated. Thus, a synchronous evaluation of the complete system represents an infeasible approach for such tasks, since it can lead to the inappropriate evaluation of intermittent singular systems.

In addition, the discrete event handling is insufficiently specified in the Modelica language definition. There is a clear lack of specification for describing what is supposed to happen exactly if one event is subsequently causing (or canceling) other events during the same point of simulation time. This concerns for example the MultiBondLib [8] and its impulse-models. The correctness of these models cannot be proven on the basis of the language-specification. Indeed, the correct simulation of these models is bound to the specific implementation in Dymola.

2.6 Tedious complexity

In the attempt to enhance the Modelica-language with regard to certain applicationspecific tasks, the original language has lost some of its original beauty and

clarity. An increasing amount of specific elements have been added to the language that come with rather small advantages. Several of these small add-ons are potential sources for problems when structural variability is concerned. Thus, a clean-up of the language is an inevitable prerequisite for any further development in this field. Furthermore the language is subverted in daily practice by foreign elements, i.e., so-called annotations.

2.7 Summary

To express structural changes, a corresponding modeling language has to meet certain requirements. The language must support discrete events and hence support hybrid modeling, since structural changes clearly represent a discrete event. Furthermore, it must be allowed to state relations between variables or sub-models in a conditional form, so that the structure can change depending on time and state. In addition, variables and sub-models should be dynamically declarable, so that the corresponding instances can be created, handled, and deleted at run time. Modelica meets these requirements only partly and provides only very limited means for the description of such models.

2.8 MOSILAB

MOSILAB[7] offers a first approach to handling variable structure systems in a more general sense. It combines an extensive subset of Modelica with a description language for state charts to handle the transition between different modeling modes. MOSILAB features the dynamic creation of sub-model instances, although it does so in a limited way. For us, the use of state charts represents a practical but limited solution. However, state charts do not integrate too well into the object-oriented and declarative framework of Modelica. Hence the complexity of the language had to be increased significantly and the beauty and clarity of the original Modelica language suffered in the process of extending the language.

3 Sol - A derivative language of Modelica

In attempting an enhancement of Modelica's capabilities with respect to variable structure systems, one arrives at the conclusion that a straight-forward extension of the language will not lead to a persistent solution. The introduction of additional dynamics inevitably violates some of the fundamental assumptions of the original language design and of its corresponding translation and simulation mechanisms.

Hence we have taken the decision to design a new language, optimized to suit the new set of demands. This language is called Sol. In the design process, we intend to maintain as much of the essence of Modelica as possible. To this end, we review the major strengths of Modelica:

- Modelica owns natural readable, intuitive syntax. Models can be understood even by outsiders, and beginners are enabled to quickly acquaint themselves with the language.
- The declarative, equation-based modeling approach enables the modeler to concentrate on what should be modeled, rather than forcing him or her to consider, how precisely the model is to be simulated.
- Modelica offers convenient object-oriented means for the organization of knowledge and type-generation. This makes large projects feasible and eases the knowledge transfer.
- The structural type-system of Modelica separates type-generation and implementation. Thus, even separate implementations can be compatible and exchangeable. The generic connection mechanism enables intuitive and convenient modeling.

3.1 Sol – A new language for variable structure systems

All those considerations of the previous sections have been taken into account for the design process of Sol. The decision to design a new language enables us to take a more radical, conceptually stronger approach. Hence, Sol attempts to be a language of low complexity that still enables a high degree of expressiveness.

Like Modelica, Sol provides means for declaring synchronous, non-causal relations between variables (i.e., equations). As an extension to Modelica, we furthermore offer a convenient way for declaring asynchronous, causal transmissions from one variable (or sub-model) to another. All of these declarations can be grouped in an almost arbitrary fashion. These groups of declarations may be activated or deactivated in accordance with conditions, events or predetermined sequences.

Unlike in Modelica, also the declaration of variables and sub-models can occur at the beginning of each group or subgroup. Since these groups can be stated in a conditional form, variables and sub-models may be dynamically created and deleted at run time. Hence instance creation and deletion does not need to be stated in the (typical) imperative form. It results

from the activation and deactivation of declarative groups. The dynamically created objects can be handled in an unambiguous way by the declaration of asynchronous transmissions. Identifiers can also link dynamically to an instance.

Hence systems that are expressed in Sol are described in a constructive way, where the path of construction and the corresponding interrelations might change in dependence on the current system's state or on current evaluations. Conditional declarations enable a high degree of variability in structure. The constructive approach avoids memory leaks and the description of error-prone update-processes.

The new language will be well-structured, easily readable, and intuitive to understand. The language will provide various object-oriented tools that enable the efficient handling of complex systems. The syntax and grammar of Sol is significantly stricter than the grammar of Modelica. Alternative writings have been discarded and the different sections of a model must obey a given order. This strictness unifies the writing and intends to guide towards a clear and understandable modeling style.

3.2 Example

Without going into the details concerning Sol's grammar and semantics, we provide a small, introductory example to show its potential usage. Due to Sol's similarity to Modelica and its intuitive syntax, the example should be understandable in its main functionality. In addition to classic equations Sol features copy-transmission (<<) and move-transmissions (<-). We model a simple machine, consisting of an engine that drives a fly-wheel. Two models are provided for the engine: The first model "Engine1" applies a constant torque on the flange. In the second model "Engine2", the torque is dependent on the positional state similar to a piston-engine. The machine-model connects the engine and the fly-wheel. It contains a structural change that is reflected by a substitution of the engine-models. Initially, the fly-wheel is at rest, and the more complex engine model is used. When the speed exceeds a certain threshold, it seems appropriate to average the torque. Thus, the simpler engine-model is used instead.

The structural change is contained in the model `Machine`. It declares a Boolean state-variable `fast` that determines which model to use. Please note, that the conditional if-clauses also contain declarations of

```
package Rotational
connector Flange
   interface:
      static potential Real phi;
      static flow Real t;
end flange;
partial model Engine
   interface:
      parameter Real meanTorque << 1;
      static Flange f;
end Engine;
model Engine1 extends Engine;
   implementation:
      f.t = meapnTorque;
end Engine1;
model Engine2 extends Engine;
   implementation:
      static Real transmission;
      transmission = 1+sin(f.phi);
      f.t = meanTorque*transmission;
end Engine2;
model FlyWheel
   interface:
      parameter Real inertia << 1;
      static Flange f;
      static Real w;
   implementation:
      static Real z;
      w = der(f.phi);
      z = der(w);
      f.t = inertia*z;
      when initial
            then w=0; f.phi=0; end;
end FlyWheel;
model Machine
 implementation:
   static FlyWheel Wheel1{inertia << 10};
   static Boolean fast;
   if fast then
     static Engine1 E{meanTorque << 100};
     connection(E.f,Wheel1.f);
   else then
     static Engine2 E{meanTorque << 100};
     connection(E.f,Wheel1.f);
   end;

   when initial then fast << false; end;
   when Wheel1.w > 50
          then fast << true; end;
end Machine;
end Rotational;
```

sub-models. This enables a convenient, easily readable formulation of the structural change based on the current system state. There is also no need for an explicit model of the transition or manual disconnections.

The example code below presents an alternative solution for the machine-model. The identifier `E` is declared to be *dynamic*. This means: It can be dynamically linked to any model-instance that is type-compatible with `Engine`. The corresponding instances are simply declared anonymously in the conditional when-clauses. The type of a model is solely defined by its interface-section.

```
1 model Machine
2  implementation:
```

```
 3    static FlyWheel Wheel1{inertia << 10};
 4    dynamic Engine E;
 5    connection(E.f,Wheel1.f);
 6    when initial then
 7       E <- Engine2{meanTorque << 100};
 8    end;
 9    when Wheel1.w > 50 then
10       E <- Engine1{meanTorque << 100};
11    end;
12 end Machine;
```

This simple example contains only a very simple structural change that is basically reflected by the replacement of a single equation. Hence this could have also been modeled in Modelica, but not at this level of abstraction. The complete replacement of a model, as it is done here, can as well be used for more elaborate multi-level models.

4 Implementation and on-going development

A first version of the language definition of Sol has been written down in the form of an internal report. It forms the fundamentals for a corresponding implementation that is currently under development. This implementation will be represented by an interpreter that parses the model-file, instantiates a selected model and starts simulation. In addition to its main task, the interpreter will provide various tools for the analysis of the object-hierarchy, type-structure, etc.

Whereas the pair of a compiler and a simulator is the preferred choice for high-end simulation tasks, an interpreter is an appropriate tool for research work on language design. The development process becomes much easier, faster and more flexible. Hence the development of the interpreter can proceed in parallel with a further refinement of the language. Also, new debugging techniques will be needed that can be better provided by an interpreter, since all necessary meta-information is available. Of course, any interpreter (even if it is well written) suffers from a certain computational overhead that may prevent its usage for highly demanding simulation applications. Hence an important aspect will be to sketch the development of a corresponding compiler.

4.1 Future goals

Sol is a language primarily conceived for research purposes. We want to explore the full power of a declarative modeling approach and how it can handle potential, future problem fields. Some of our goals and motivations are similar to [6], although we are coming from a different direction. The implementation of Sol will be a small and open project that should enable other researchers to test and validate their ideas with a moderate effort. The longer term goal of our research is to significantly extend Modelica's expressiveness and range of application. Furthermore, the Sol-project gives us a development-platform for technical solutions that concerns the handling of structurally changing equation systems. This includes solutions for dynamic recausalization or the dynamic handling of structural singularities.

It is not our target to immediately change the Modelica standard or to establish an alternative modeling language. Our scientific work is intended to merely offer suggestions and guidance for future development. This will primarily benefit future development of Modelica, but our results may also prove useful to other modeling communities and researchers.

5 Conclusion

The development of a new modeling language should be a well considered step, since it incorporates a lot of effort. This does not only concern the developers of the language and the corresponding software, it includes as well the potential modelers and users that are expected to get themselves acquainted with the new methodology. However, the continuous progress of modeling technology generates a new set of demands. This makes such a step finally inevitable.

In this workshop-paper, we offered a first glance of Sol, our new modeling language. Sol has been designed to enable the modeling of variable-structure systems using an equation-based framework. While its development is currently still at the beginning, we expect to make significant progress in the near future. In the longer term, we hope that our research will benefit Modelica's future development (cf. [9]).

References

[1] Broman, D., Fritzson, P., Furic, S.: *Types in the Modelica Language*. In: Proceedings of the Fifth International Modelica Conference, Vienna, Austria (2006) Vol. 1, 303-315

[2] Cellier, F.E., Krebs, M.: *Analysis and Simulation of Variable Structure Systems Using Bond Graphs and Inline Integration*. In: Proc. ICBGM'07, 8th SCS Intl. Conf. on Bond Graph Modeling and Simulation, San Diego, CA, (2007) 29-34.

[3] Elmqvist, H., Otter, M., Díaz López, D.: *Collision Handling for the Modelica MultiBody Library*. In: Proc. 4th International Modelica Conference, Hamburg, Germany (2006) 45-53

[4] Höpler, R., Otter, M.: *A Versatile C++ Toolbox for Model Based, Real Time Control Systems of Robotic Manipulators*. In: Proc. of 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, Maui, USA, (2001) 2208-2214

[5] Otter, M., Elmqvist, H., Mattsson, S.E.: *Hybrid Modeling in Modelica Based on the Synchronous Data Flow Principle*. In: Proc. IEEE International Symposium on Computer Aided Control System Design, Hawaii. (1999) 151-157

[6] Nilsson, H., Peterson J., Hudak, P.: *Functional Hybrid Modeling*. In: Proceedings of the 5th International Workshop on Practical Aspects of Declarative Languages, New Orleans, LA (2003) 376—390

[7] Nytsch-Geusen, C. et. al.: *Advanced modeling and simulation techniques in MOSILAB: A system development case study*. In: Proceedings of the Fifth International Modelica Conference, Vienna, Austria (2006) Vol. 1, 63-71

[8] Zimmer, D., Cellier, F.E.: *The Modelica Multi-bond Graph Library*. In: Proc. 5th Intl. Modelica Conference, Vienna, Austria (2006) Vol. 2, 559-568

[9] Zimmer, D.: *Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems*. In: Proc. 6th International Modelica Conference, Bielefeld, Germany (2008), Vol.1 pp. 47-56

Corresponding author: Dirk Zimmer
Institute of Computational Science, ETH Zürich
CH-8092 Zürich, Switzerland
dzimmer@inf.ethz.ch

Accepted EOOLT 2007, June 2007
Received: August 15, 2007
Accepted: August 25, 2007

Dear Readers,

We are glad to continue the SNE Special Issue Series with this special issue SNE 17/2 on 'Object-oriented and Structural-dynamic Modelling and Simulation I'. The editorial policy of SNE Special Issues is to publish high quality scientific and technical papers concentrating on state-of-the-art and state-of-research in specific modelling and simulation oriented topics in Europe, and interesting papers from the world wide modelling and simulation community. The subject 'Object-oriented and Structural-dynamic Modelling and Simulation' fulfils all prerequisites for a special issue, and we are happy, that the organisers of EOOLT 2007 workshop and of EUROSIM 2007 special session on structural-dynamic systems agreed to select contributions from these events to be published in revised and/or extended form in an SNE special issue. The subject indeed turned out to be of big interest, and more contributions became candidates than could be published into one special issue- so a second special issue 'Object-oriented and Structural-dynamic Modelling and Simulation II' is planned for 2008 or 2009. The already announced SNE Special Issue 'Verification and Validation' is postponed to 2008 or 2009.

I would like to thank all authors and all people who helped in managing this SNE Special Issue, especially the Guest Editors, Peter Fritzson (Linköping University, Sweden), François Cellier (ETH Zurich, Switzerland), Christoph Nytsch-Geusen, Fraunhofer FIRST, Berlin, Germany), Peter Schwarz (Fraunhofer EAS, Dresden, Germany), and Borut Zupancic (Univ. Ljubljana, Slovenia).

Felix Breitenecker, Editor-in-Chief SNE; Felix.Breitenecker@tuwien.ac.at

SNE - Editors /ARGESIM
c/o Inst. f. Analysis and Scientific Computation
Vienna University of Technology
Wiedner Hauptstrasse 8-10, 1040 Vienna, AUSTRIA
Tel + 43 - 1- 58801-10115 or 11455, Fax – 42098
sne@argesim.org; www.argesim.org

Editorial Info – Impressum

SNE Simulation News Europe ISSN 1015-8685 (0929-2268).

Scope: Technical Notes and Short Notes on developments in modelling and simulation in various areas /application and theory) and on bechmarks for modelling and simulation, membership information for EUROSIM and Simulation Societies.

Editor-in-Chief: Felix Breitenecker, Inst. f. Analysis and Scientific Computing, Vienna University of Technology, Wiedner Hauptstrasse 8-10, 1040 Vienna, Austria; Felix.Breitenecker@tuwien.ac.at

Layout: Markus Wallerberger, ARGESIM TU Vienna; markus.wallerberger@gmx.at

Printed by: Grafisches Zentrum, TU Vienna, Wiedner Hauptstrasse 8-10, 1040, Vienna, Austria

Publisher: ARGESIM/ASIM; ARGESIM, c/o Inst. for Scientific Computation, TU Vienna, Wiedner Hauptstrasse 8-10, 1040 Vienna, Austria, and ASIM (German Simulation Society), c/o Wohlfartstr. 21b, 80939 Munich

TN

Functional Hybrid Modeling from an Object-Oriented Perspective

Henrik Nilsson, University of Nottingham, United Kingdom, *nhn@cs.nott.ac.uk*
John Peterson, Western State College, USA, *jpeterson@western.edu*
Paul Hudak, Yale University, USA, *paul.hudak@yale.edu*

Declaration: This paper is closely on [19] that was published in the Proceedings of Practical Aspects of Declarative Languages (PADL) 2003. The paper has been updated and adapted for the Equation-Based Object-Oriented Languages and Tools (EOOLT) 2007 Workshop.

The modeling and simulation of physical systems is of key importance in many areas of science and engineering, and thus can benefit from high-quality software tools. In previous research we have demonstrated how *functional programming* can form the basis of an expressive language for *causal* hybrid modeling and simulation. There is a growing realization, however, that a move toward *non-causal* modeling is necessary for coping with the ever increasing size and complexity of modelling problems. Our goal is to combine the strengths of functional programming and non-causal modeling to create a powerful, strongly typed *fully declarative modeling language* that provides modeling and simulation capabilities beyond the current state of the art: in particular, support for highly structurally dynamic systems. Additionally, we think our approach could serve as a semantical framework for studying modeling and simulation languages supporting structural dynamism, and maybe even as a core language in systems where the surface syntax is more conventional. Although our work is still in its very early stages, we believe that this paper clearly articulates the need for improved modeling languages and shows how functional programming techniques can play a pivotal role in meeting this need.

SNE 17/2, September 2007

Introduction

Modeling and *simulation* is playing an increasingly important role in the design, analysis, and implementation of real-world systems. In particular, whereas modeling fragments of systems in isolation was deemed sufficient in the past, considering the interaction of these fragments *as a whole* is now necessary. The resulting models are large and complex, and span multiple physical domains.

Furthermore, these models are almost invariably *hybrid*: they exhibit both continuous-time and discrete-time behaviors. In fact, the very structure of the modeled system changes over time. Such models are known as *structurally dynamic*. In general, the total number of structural configurations, or modes, can be enormous, or even unbounded. We refer to systems whose number of modes cannot be practically predetermined as *highly structurally dynamic*. While supporting structural dynamism is hard, supporting highly structurally dynamic systems is even harder as this necessitates comprehensive and flexible solutions of a number of important subproblems: see Sect. 4.

There are two broad language categories of modeling and simulation languages. *Causal* (or *block-oriented*) languages are most popular; languages such as Simulink and Ptolemy II [13] represent this style of modeling. In causal modeling, the equations that represent the physics of the system must be written so that the direction of signal flow, the causality, is explicit. The second, but less populated, class of language is *non-causal*, where the model focuses on the interconnection of the components of the system being modeled, from which causality is then inferred. Such languages often support an *object-oriented* approach to modeling. Examples include Dymola [5] and Modelica [15].

The main drawback of causal languages is the need to explicitly specify the causality. This hampers modularity and reuse [2]. Non-causal languages address this problem by allowing the user to describe a model in a way which does not commit to any specific causality. The appropriate causality constraints are then inferred using both symbolic and numerical methods depending on how the model is being used. Unfortunately, current non-causal modeling languages tend to sacrifice generality when it comes to hybrid modeling: in particular, we are not aware of any *declarative* non-causal modeling language that supports highly structurally dynamic models, even if recent efforts like MOSILAB [20] and Sol [28] are important steps in that direction.

In previous research at Yale, we have developed a framework called *Functional Reactive Programming* (FRP) [26], which is suited for causal hybrid modeling. This framework is embodied in a language called *Yampa* (see `haskell.org/yampa`) as an extension of Haskell. Yampa permits highly structurally dynamic hybrid systems to be described clearly and concisely [18], at present, however, Yampa lacks integration with sophisticated numerical solvers, and its applicability for serious simulation work is thus limited. In addition, because the full power of a functional language is available, it exhibits a high degree of modularity, allowing reuse of components and design patterns. It also employs Haskell's polymorphic type system to ensure that signals are connected consistently, even as the system topology changes. The semantic foundations of Yampa are well defined and understood, making models expressed using Yampa suited for formal manipulation and reasoning. Yampa and its predecessors have been used in robotics simulation and control as well as a number of related domains [23, 24]. It has even been used for video games [4, 3]. We are currently investigating biological cell population modeling, where Yampa's support for highly structurally dynamic systems provides an interesting declarative approach to handling cell division in contrast to the imperative approach of agent-based simulators [12].

Non-causal modeling and FRP complement each other almost perfectly. We therefore aim to integrate the core ideas of FRP with non-causal modeling to create *Hydra*, a powerful, fully declarative modeling language combining the strengths of each. If we treat causality and dynamism as two dimensions in the modeling language design space, we see that Hydra occupies a unique point:

	Mostly static structure	Highly dynamic structure
Causal	Simulink	Yampa
Non-causal	Modelica	Hydra

MOSILAB and Sol are somewhere between Modelica and Hydra.

We refer to the combined paradigm of functional programming and non-causal, hybrid modeling as *Functional Hybrid Modeling*, or FHM. Conceptually, FHM can be seen as a generalization of FRP, since FRP's *functions* on signals are a special case of FHM's *relations* on signals. In its full generality, FHM, like FRP, also allows the description of structurally dynamic models.

The main contribution of this paper is that it outlines how notions appropriate for non-causal, hybrid simulation in the form of *first-class relations on signals* and *switch constructs* can be integrated into a functional language, yielding a non-causal modeling language supporting structural dynamism. It also identifies key research issues, and suggests how recent developments in the field of programming languages could be employed to address those issues.

1 Yampa

To help readers who are not familiar with Functional Reactive Programming put the ideas of this paper into context, we provide a brief review of the key ideas of Yampa in the following. For further details, see earlier papers on Yampa [9, 18]

1.1 Fundamental concepts

Yampa is based on two central concepts: *signals* and *signal functions*. A signal is a function from time to a value:

$$Signal\ \alpha \approx Time \to \alpha$$

Time is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, the type of a varying electrical voltage might be *Signal Voltage*.

A *signal function* is a function from *Signal* to *Signal*:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \to Signal\ \beta$$

When a value of type $SF\ \alpha\ \beta$ is applied to an input signal of type $Signal\ \alpha$, it produces an output signal of type $Signal\ \beta$. Signal functions are *first classentities* in Yampa. Signals, however, are not: they only exist indirectly through the notion of signal function. In general, the output of a signal function at time t is uniquely determined by the input signal on the interval $[0,t]$. If a signal unction is such that the output at time t only depends on the input at the very same time instant t, it is called *stateless*. Otherwise it is *stateful*.

1.2 Composing signal functions

Programming in Yampa consists of defining signal functions compositionally using Yampa's library of primitive signal functions and a set of combinators. Yampa's signal functions are an instance of the arrow framework proposed by Hughes [10]. Three combinators from that framework are *arr*, which lifts an ordi-

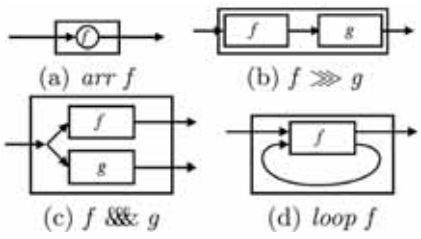


Figure 1. Basic signal function combinators

nary function to a stateless signal function, and the two signal function composition combinators ⋘ and &&& :

$$arr \quad :: (a \to b) \to SF\,ab$$
$$(\lll) \;:: SF\,bc \to SF\,ab \to SF\,ac$$
$$(\&\&\&) \;:: SF\,ab \to SF\,ac \to SF\,a\,(b,c)$$

We can think of signals and signal functions using a simple flow chart analogy. Boxes represent signal functions, with one signal flowing in to the box's input port and another signal flowing out of the box's output port. Figure 1 illustrates some of the central arrow combinators using this analogy. The similarity to a block-oriented modeling language like Simulink is hopefully clear. The main difference is that the notion of composing blocks into larger blocks has been formalized through a handful of composition combinators, which is helpful from a semantical perspective, in contrast to the more unstructured approach of connecting outputs to inputs in an arbitrary fashion.

1.3 Arrow syntax

While the arrow framework provides a useful semantical structure, it is not convenient for expressing large networks. It is much easier to simply connect whatever needs to be connected Simulink style, e.g. by naming nodes and then explicitly stating the connection topology. Fortunately, it is easy to provide a layer of syntax that allows this, and then translate this into a network description in terms of the core arrow combinators. Paterson's arrow notation [22] does exactly that. An expression denoting a signal function has the form:

proc $pat \to$ do
  $pat_1 \leftarrow sfexp_1 \prec exp_1$
  $pat_2 \leftarrow sfexp_2 \prec exp_2$
  …
  $pat_n \leftarrow sfexp_n \prec exp_n$
  $return\,A \prec exp$

The keyword **proc** is analogous to the λ in λ-expressions, pat and pat_i are patterns binding signal variables pointwise by matching on instantaneous signal values, exp and exp_i are expressions defining instantaneous signal values, and exp_i are expressions denoting signal functions. The idea is that the signal being defined pointwise by each exp_i is fed into the corresponding signal function $sfexp_i$, whose output is bound pointwise in pat_i. The overall input to the signal function denoted by the **proc**-expression is bound by pat, and its output signal is defined by the expression exp. The signal variables bound in the patterns may occur in the signal value expressions, but not in the signal function expressions $sfexp_i$). An optional keyword **rec**, applied to a group of definitions, permits signal variables to occur in expressions that textually precede the definition of the variable, allowing recursive definitions (feedback loops).

For a concrete example, consider the following:

sf = **proc** $(a,b) \to$ **do**
  $(c1,c2) \leftarrow sf1$ &&& $sf2 \prec a$
  $d \leftarrow sf3 \lll sf4 \prec (c1,b)$
  rec
    $e \leftarrow sf5 \prec (c2,d,e)$
  $return\,A \prec (d,e)$

Note the use of the tuple pattern for splitting sf's input into two "named signals", a and b. Also note the use of tuple expressions and patterns for pairing and splitting signals in the body of the definition; for example, for splitting the output from $sf1$ &&& $sf2$. Also note how the arrow notation may be freely mixed with the use of basic arrow combinators.

1.4 Accessing the Environment

While some aspects of a program are naturally modeled as continuous signals, other aspects are more naturally modeled as *discrete events*. To this end, Yampa introduces the *Event* type, isomorphic to Haskell's *Maybe* type:

data *Event a = NoEvent | Event a*

The instantaneous value of signal of type *Event T* for some type *T* is either *NoEvent* or *Event x* for some value *x* of type *T*, thus mimicking a discrete-time signal that is only defined at discrete points in time.

$$switch :: SF\,a\,(b, Event\,c) \to (c \to SF\,ab) \to SF\,ab$$

The *switch* combinator switches from one subordinate

signal function into another when a switching event occurs. Its first argument is the signal function that initially is active. It outputs a pair of signals. The first defines the overall output while the initial signal function is active. The second signal carries the event that will cause the switch to take place. Once the switching event occurs, *switch* applies its second argument to the value of the event and switches into the resulting signal function.

Thus, note that the second argument of *switch* is a *function* of type $c \to SF\ ab$, that, when given the value of type c carried by the event, *dynamically* computes a new signal function to switch into. Using a Simulink analogy, *switch* in principle rips out a block, and then dynamically instantiates a parameterized block as a replacement. The design of *switch* thus exploits the fact that signal functions ("blocks") are first class entities in Yampa.

Yampa also includes *parallel* switching constructs that maintain *dynamic collections* of signal functions connected in parallel [18]. Signal functions can be added to or removed from such a collection at runtime in response to events, while *preserving* any internal state of all other signal functions in the collection; see Fig. 2. The first class status of signal functions in combination with switching over dynamic collections of signal functions makes Yampa an unusually flexible language for describing hybrid systems. For example, this makes it possible to handle systems where the number of modeled entities varies over time, like cell population models as mentioned earlier (see Introduction).

2 Non-causal and hybrid modeling

While the simulation of pure continuous systems is relatively well understood, hybrid systems pose a number of unique challenges [16, 1]. Problems include handling a large number of modes, event detection, and consistent initialization of state variables. The integration of hybrid modeling with non-causal modelling raises further problems. Indeed, current

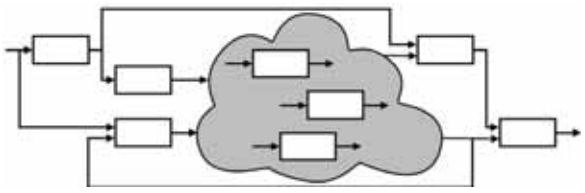
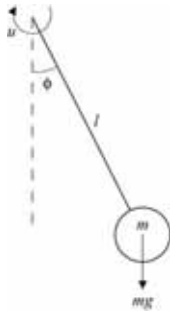


Figure 2. System of interconnected signal functions with varying structure



(a) Pendulum

```
model BreakingPendulum
  parameter Real m=1, g=9.81, L=0.5;
  parameter Boolean Broken;
  input Real u;
  Real pos[2], vel[2];
  Real phi(start=PI/4), phid;
equation
  vel = der(pos);
  if not Broken then
     // Equations of the pendulum
     pos = {L*sin(phi), -L*cos(phi)};
     phid = der(phi);
     m*L*L*der(phid) + m*g*L*sin(phi)=u;
  else
     // Equations of the free-flying mass
     m*der(vel) = m*{0, -g};
  end if;
end BreakingPendulum;
```

(b) Modelica model

Figure 3. A pendulum, subject to externally applied torque and gravity.

non-causal modeling languages are quite limited in their ability to express hybrid systems. Many of the limitations are related to the symbolic and numerical methods that must be used in the non-causal approach. But another important reason is that most such systems insist on performing all symbolic manipulations *before* simulation begins [16]. Avoiding these limitations is an important part of our approach, see Sec. 4.

Since Modelica is representative of state-of-the-art, non-causal, hybrid modeling languages, we illustrate the limitations of present languages with an example from the Modelica documentation [14, pp. 31–33]. The system is a pendulum in the form of a mass *m* at the end of a rigid, massless rod, subject to gravity *mg* and an externally applied torque *u* at the point of suspension; see Fig. 3(a). Additionally, the rod could break at some point, causing the mass to fall freely.

Figure 3(b) shows a Modelica model of this system that, on the surface, looks like it achieves the desired result. Note that it has two modes, described by conditional equations. In the non-broken mode, the position `pos` and velocity `vel` of the mass are calculated from the state variables `phi` and `phid`. In the broken mode, `pos` and `vel` become the new state variables. This implies that state information has to be transferred between the non-broken and broken mode. Furthermore, the causality of the system is different in the two modes. When non-broken, the equation relating `vel` and `pos` is used to compute `vel` from `pos`. When broken, the situation is reversed.

These facts make simulation hard. Modelica attempts to simplify matters by avoiding too radical structural

changes. To that end, Modelica either requires the condition for selecting between two sets of equation to be a *parameter*, and thus unchanging during simulation, or else that the number of equations in each set is the same. In this case, as the number of equations is not the same, `Broken` has to be declared a parameter. Therefore the model above does not really solve the hybrid simulation problem at all! In order to actually model a pendulum that dynamically breaks at some point in time, the model must be expressed in some other way. The Modelica documentation suggests a causal, block-oriented formulation with explicit state transfer. Unsurprisingly, the result is considerably more verbose, nullifying the advantage of working in a non-causal language.

Moreover, even if `Broken` were allowed to be a dynamic variable, a fundamental problem would remain: once the pendulum has broken, it cannot become whole again. Modelica provides no way to declaratively express the *irreversibility* of this structural change. The best that can be done is to capture this fact indirectly through a state machine model that control the value of `Broken`.

3 Integrating functional programming and non-causal modeling

In the preceding sections we discussed the advantages of non-causalmodeling and the importance of hybrid modeling. We also pointed out serious shortcomings in current modeling languages with respect to these features. In this section, we describe a new way to combine non-causal and hybrid modeling techniques that addresses these issues. Inspired by FRP and Yampa, the two key ideas are to give first-class status to relations on signals and to provide constructs for discrete switching between relations. The result is Hydra, a functional hybrid modelling language capable of representing structurally dynamic systems.

While we, based on our experience of Yampa, believe that a language like Hydra would be a very expressive and powerful modeling and simulation language in its own right, we would like to emphasize that we also think our approach could serve as a valuable semantical framework for general study of modelling and simulation languages that supports structural dynamism, and maybe even as a core language in systems where the surface syntax is more conventional. Thus, what is important in the following is not the syntax (which is tentative and likely lacking in many ways), but the underlying principles.

3.1 First-class signal relations

A natural mathematical description of a continuous signal function is that of an ODE in explicit form. A function is just a special case of the more general concept of a *relation*. While functions usually are given a causal interpretation, relations are inherently non-causal. Differential Algebraic Equations (DAEs), which are at the heart of non-causal modeling, express dependences among signals without imposing a causality on the signals in the relation. Thus it is natural to view the meaning of a DAE as a non-causal *signal relation*, just as the meaning of an ODE in explicit form can be seen as a causal signal function. Since signal functions and signal relations are closely connected, this view offers a clean way of integrating non-causal modeling into an Yampa-like setting.

In the following, first-class signal relations are made concrete by proposing a (tentative) system for integrating them into a polymorphically typed functional language. Signal functions are also useful, but since they are just relations with explicit causality, we need not consider them in detail in the following.

Similarly to the signal function type *SF* of Yampa (Sect. 2.1), we introduce the type *SR α* for a relation on a signal of type *Signal α*. Specific relations use a more refined type; e.g., for the derivative relation *der* we have the typing:

der :: *SR* (*Real*, *Real*)

Since a signal carrying pairs is isomorphic to a pair of signals, we can understand *der* as a binary relation on two real-valued signals.

Next we need a notation for defining relations. Inspired by the arrow notation (Sect. 1.3), we introduce the following to denote a signal relation:

sigrel *pattern* **where** *equations*

The pattern introduces *signal variables* that at each point in time are bound to the *instantaneous* value of the corresponding signal. Given *p* :: *t*, we have:

sigrel *p* **where** … :: *SR t*

Consequently, the equations express relationships between instantaneous signal values. This resembles the standard notation for differential equations in mathematics. For example, consider $x' = f(y)$, which means that the instantaneous value of the derivative of (the signal) x at every time instant is equal to the value obtained by applying the function f to the instantaneous value of y.

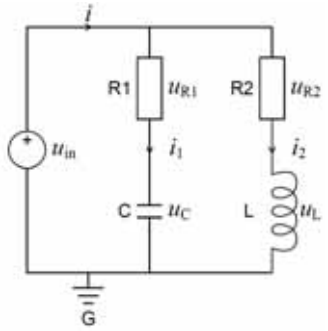


Figure 4. A simple electrical circuit

We introduce two styles of equations:

$$e_1 = e_2$$
$$sr \lozenge e_3$$

where e_i are expressions (possibly introducing new signal variables), and sr is an expression denoting a signal relation. We require equations to be well-typed. Given $e_i :: t_i$, this is the case iff $t_1 = t_2$ and $sr :: SR\,t_3$.

The first kind of equation requires the values of the two expressions to be equal at all points in time. For example:

$$f(x) = g(y)$$

where f and g are functions.

The second kind allows an arbitrary relation to be used to enforce a relationship between signals. The symbol $\lozenge$ can be thought of as relation application; the result is a constraint which must hold at all times. The first kind of equation is just a special case of the second in that it can be seen as the application of the identity relation.

For another example, consider a differential equation like $x' = f(x, y)$. Using our notation, this equation could be written:

$$der \lozenge (x, f(x, y))$$

where *der* is the relation relating a signal to its derivative. For convenience, a notation closer to the mathematical tradition should be supported as well:

$$\mathbf{der}(x) = f(x, y)$$

The meaning is exactly as in the first version.

We illustrate our language by modeling the electrical circuit in Figure 4 (adapted from [14]). The type *Pin* is a record type describing an electrical connection. It has fields *v* for voltage and *i* for current (the name *Pin* is perhaps a bit misleading since it just represents a pair of physical quantities, *not* a physical "pin component"; i.e., *Pin* is the type of *signal variables* rather than *signal relations*).

$$twoPin :: SR(Pin, Pin, Voltage)$$
$$twoPin = \mathbf{sigrel}(p, n, u)\ \mathbf{where}$$
$$u = p.v - n.v$$
$$p.i + n.i = 0$$

$$resistor :: Resistance \to SR(Pin, Pin)$$
$$resistor(r) = \mathbf{sigrel}(p, n)\ \mathbf{where}$$
$$twoPin \lozenge (p, n, u)$$
$$r \cdot p.i = u$$

$$inductor :: Inductance \to SR(Pin, Pin)$$
$$inductor(l) = \mathbf{sigrel}(p, n)\ \mathbf{where}$$
$$twoPin \lozenge (p, n, u)$$
$$l \cdot der(p.i) = u$$

$$capactior :: Capacitance \to SR(Pin, Pin)$$
$$capacitor(l) = \mathbf{sigrel}(p, n)\ \mathbf{where}$$
$$twoPin \lozenge (p, n, u)$$
$$c \cdot der(u) = p.i$$

As in Modelica, the resistor, inductor and capacitor models are defined as extensions of the *twoPin* model. However, we accomplish this directly with functional abstraction rather than the Modelica class concept. Note how parameterized models are defined through functions *returning* relations. Since the parameters are normal function arguments, *not* signal variables, their values remain unchanged throughout the lifetime of the returned relations (compare to Modelica's parameter-variables mentioned in Sect. 2).

To assemble these components into the full model, we adopt a Modelica-like **connect**-notation as a convenient abbreviation for connection equations. This is syntactic sugar which is expanded to proper connection equations, i.e. equality constraints or sum-to-zero equations depending on what kind of physical quantity is being connected. We assume that a voltage source model *vSourceAC* and a ground model *ground* are available in addition to the component models defined above. Moreover, we are only interested in the total current through the circuit, and, as there are no inputs, the model thus becomes a *unary* relation:

$$simpleCircuit :: SR\ Current$$
$$simpleCircuit = \mathbf{sigrel}\ \mathrm{i}\ \mathbf{where}$$
$$resistor\ (1000) \lozenge (r1p, r1n)$$

resistor (2200) ◊ (*r2p*, *r2n*)
capacitor (0.00047) ◊ (*cp*, *cn*)
inductor (0.01) ◊ (*lp*, *ln*)
vSourceAC(12) ◊ (*acp*, *acn*)
ground ◊ *gp*
connect *acp*, *r1p*, *r2p*
connect *r1n*, *cp*
connect *r2n*, *lp*
connect *acn*, *cn*, *ln*, *gp*
i = *r1p.i* + *r2p.i*

3.2 Modeling systems with dynamic structure

In order to describe structurally dynamic systems we need to represent an evolving structure. To this end, we introduce two Yampa-inspired switching conallows repeated switching between equation groups. structs: the *recurring switch* and the *progressing switch*. The recurring switch allows repeated switching between equation groups. In contrast, the progressing switch expresses that one group of equations *first* is in force, and then, *once* the switching condition has been fulfilled, another group, thus irreversibly progressing to a new structural configuration. For either sort of switching, difficult issues such as state transfer and proper initialization have to be considered.

We revisit the breaking pendulum example from Sect. 3 to illustrate these switching constructs. To deal with initialization and state transfer, we introduce special initialization equations that are only active at the time of switching, that is, during *events*, and we allow such equations to refer to the values of signal variables just prior to the event through a special **pre**-construct devised for that purpose. The initialization equations describe the initial conditions of the DAE after a switch. Mathematically, these equations must yield an initial value for every state variable in the new continuous equations. It is important that each branch of a switch can be associated with its own initialization equations, since each such branch may introduce its proper set of state variables. Initialization equations typically state continuity assumptions, like *pos* and *vel* below.

First, consider a direct transliteration of the equation part of the Modelica model using a recurring switch. The necessary initialization equations have also been added:

vel = **der**(*pos*)
switch *broken*
  when *False* **then**
    init *phi* = *pi*/4
    init *phid* = 0
    pos = {*l* · sin (*phi*),−*l* · cos (*phi*)}
    phid = **der**(*phi*)
    m · *l* · *l* · **der**(*phid*) + *m* · *g* · *l* · sin (*phi*) = *u*
  when *True* **then**
    init *vel* = **pre**(*vel*)
    init *pos* = **pre**(*pos*)
    m · **der**(*vel*) = *m* · {0,−*g*}

A recurring switch has one or more **when**-branches. The idea is that the equations in a **when**-branch are in force whenever the pattern after **when** (which may bind variables) matches the value of the expression after **switch**. Thus, whenever that value changes, we have an event and a switch occurs (this is similar to **case** in a functional language).

To express the fact that the pendulum cannot become whole once it has broken, we refine the model by changing to a progressing switch:

vel = **der**(*pos*)
switch *broken*
  first
    …
  once *True* **then**
    …

A progressing switch has one **first**-branch and one or more **once**-branches Initially, the equations in the **first**-branch are in force, but as soon as the value of the expression after **switch** matches one of the **once**-patterns, a switch occurs to the equations in the corresponding branch, after which no further switching occurs (for that particular instance of the switch).

By combining recursively-defined relations and progressing switches, it is possible to express very general sequences of structural changes over time, from simple mode transitions to making and breaking of connections between objects. A simple example of a recursively defined relation parameterized on a discrete state variable *n* is shown below. Initially, the relation behaves according to the equations in the **first**-branch, which may depend on *n*. Whenever the

switching condition is fulfilled, the relation switches to a new instance of itself with the parameter n increased by one. In functional parlance, this is a form of tail call.

```
sysWithCntr :: Int → SR (Real, Real)
sysWithCntr (n) = sigrel (x, y) where
    switch …
        first
            …
        once … then
            sysWithCntr (n + 1) ◊ (x, y)
```

As explained in Sect. 2.5, Yampa supports even more radical structural changes, including dynamic addition and deletion of objects. Our goal is to carry over as much as possible of that functionality to Hydra.

4 Implementation issues

There are a number of challenges that must be addressed in an implementation of a language like Hydra. The primary issues are ensuring model correctness, simulation in the presence of dynamic mode changes, and mode initialization.

It is critical that dynamic changes in the model should should not weaken the static checking of the model, i.e. we want to ensure *compositional correctness*. A Haskell-like polymorphic type system, as in Yampa, ensures that the system integrity is preserved. In addition we would like to find at least necessary conditions for statically ensuring that causality analysis can always be carried out, that the equations at least could have a solution, and so on, regardless of how relations are composed dynamically. An example of a necessary but not sufficient condition is that the number of equations and number of variables agree, and that each variable can be paired with one equation. Since it will be necessary to keep track of the balance between equations and variables across relation boundaries, it is natural to integrate this aspect into the type system. Similar considerations apply to the number of initialization equations and continuous state variables. Recent work on dependent types is relevant here [27]. We also aim at extending the type system to handle physical dimensions [11].

In a highly structurally dynamic language, it is impossible to identify all possible operating modes and then factor them out as separate systems. Modes thus have to be generated *dynamically* during simulation as follows. Whenever a switch occurs, a new, global, "flattened" DAE has to be generated. The DAE is obtained by first carrying out the necessary discrete processing, which amounts to standard functional evaluation, including evaluation of the *relational expressions* in the equations that are to be active after the switch. The evaluation of relational expression is what creates *new instances* of relations, and carrying out the instantiation dynamically when switching occurs is what enables modelling of highly structurally dynamic systems. Once the new flattened DAE has been generated, it is subjected to causality analysis and other symbolic manipulations in preparation for simulation using suitable numerical methods [21, 6, 7]. The result is causal simulation code.

The hybrid bond graph simulator HYBRSIM has demonstrated the feasibility of this dynamic approach, and that it indeed allows some difficult cases to be handled [17]. However, HYBRSIM is an *interpreted* system. Simulation is thus slowed down both by occasional symbolic processing and by the interpretive overhead. To avoid interpretive overhead, we intend to leverage recent work on run-time code generation, such as 'C [8] or Cyclone [25]. We will need to adapt the sophisticated mathematical techniques used in existing non-causal modeling languages [21, 6, 7] to this setting. In part, it may be possible to do this systematically by *staging* the existing algorithms in a language like Cyclone.

The initial conditions of the (new) differential equations must be determined on transitions from one mode to another. However, arriving at consistent initial conditions is, in general, hard. Some state variables in the continuous part of the system may exhibit discontinuities at the time of switching while others will not: simply preserving the old value is not always the right solution. Structural changes could change the set of state variables, and the relationship between the new and old states may be difficult to determine. One approach is to require the modeler to provide a function that maps the old state to the new one for each possible mode transition [1]. However, the declarative formulation of non-causal models means that the simulator sometimes has a choice regarding which continuous variables should be treated as state variables. Requiring the user to provide a state mapping function is therefore not always reasonable.

A key to the success of HYBRSIM is that bond graphs are based on physical notions such as energy and energy exchange, which are subject to continuity and

conservation principles. We intend to generalize this idea by exploring the use of *declarations* for stating such principles, along the lines illustrated in Sec. 4.2. It may also be possible to infer continuity and conservation constraints automatically based on physical dimension types.

5 Related work

There has been substantial interest in supporting structural dynamism within the non-causalmodeling community recently. The most advanced effort at present is probably MOSILAB [20]. Similarly to what is proposed here, MOSILAB supports dynamic addition and deletion of behavioral objects. The switching is controlled through a form statecharts. A modern, sophisticated DAE solver, with support for computing consistent initial conditions, is used.

However, the statechart approach implies an explicit enumeration of the modes, and even if the number of modes could be large due to combinatorial effects, this rules out a Yampa-style, truly dynamic number of simulation objects, which is the ultimate goal of Hydra.

Another aspect of MOSILAB is the use of Python for various meta-modeling tasks, such as writing "experiment scripts". We think that Hydra in itself, thanks to being a general-purpose functional language with first-class signal relations and functions, should be expressive enough to mostly provide equivalent meta-modeling capabilities, all in a uniform, declarative setting, without resorting to external imperative languages.

Sol [28] is another effort to create a non-causal modeling and simulation language supporting structural dynamism. It expressively avoids the statechart approach to retain more of the declarative clarity of languages like Modelica. It is also claimed that the Sol approach to dynamism scales better. A key aspect of the Sol approach is the capability to dynamically determine model instances. This idea seems to be somewhat similar to the notion of first-class signal functions and relations in Hydra. Like MOSILAB, Sol seems to stop short of the Hydra goal of supporting systems with a dynamic number of objects.

6 Conclusions

Hybrid modeling is a domain in which the techniques of declarative programming languages have the potential to greatly advance the state of the art. The modeling community has traditionally been concerned more with the mathematics of modeling than language issues. As a result, present modeling languages do not scale in a number of ways, particularly in hybrid systems that undergo significant structural changes. Hydra uses functional programming techniques to describe dynamically changing systems in a way that preserves the non-causal structure of the system specification and allows arbitrary switching among modes, yielding expressive power beyond current non-causal modeling languages.

Although we have not completed an implementation of Hydra, this paper demonstrates our basic design approach and maps out the design landscape. We expect that further research into the links between declarative languages and hybrid modeling will produce significant advances in this field.

Acknowledgements

The authors would like to thank the anonymous EOOLT reviewers for many useful suggestions for adapting the paper to this venue.

References

[1] P. I. Barton, C. K. Lee. *Modeling, simulation, sensitivity analysis, and optimization of hybrid systems*. Submitted to ACM Transactions on Modelling and Computer Simulation: Special Issue on Multi-Paradigm Modeling, September 2001.

[2] F. E. Cellier. *Object-oriented modelling: Means for dealing with system complexity*. In Proceedings of the 15th Benelux Meeting on Systems and Control, Mierlo, The Netherlands, pages 53–64, 1996.

[3] M. H. Cheong. *Functional programming and 3D games*. BEng thesis, University of New South Wales, Sydney, Australia, November 2005.

[4] A. Courtney, H. Nilsson, J. Peterson. *The Yampa arcade*. In Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03), pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[5] H. Elmqvist, F. E. Cellier, M. Otter. *Object-oriented modeling of hybrid systems*. In Proceedings of ESS'93 European Simulation Symposium, pages xxxi–xli, Delft, The Netherlands, 1993.

[6] H. Elmqvist, M. Otter. *Methods for tearing systems of equations in object-oriented modeling*. In Proceedings of ESM'94, European Simulation Multiconference, pages 326–332, Barcelona, Spain, June 1994.

[7] H. Elmqvist, M. Otter, F. E. Cellier. *Inline integration: A new mixed symbolic/numeric approach*. In Proc. ESM'95, European Simulation Multiconference, pages xxiii–xxxiv, Prague, June 1995.

[8] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. *'C: A language for high-level, efficient, and machine-independent dynamic code generation*. In Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96), pages 131–144, January 1996.

[9] P. Hudak, A. Courtney, H. Nilsson, J. Peterson. *Arrows, robots, and functional reactive programming*. In J. Jeuring and S. P. Jones, eds., Advanced Functional Programming, 4th International School 2002, volume 2638 of Lecture Notes in Computer Science, pp. 159–187. Springer-Verlag, 2003.

[10] J. Hughes. *Generalising monads to arrows*. Science of Computer Programming, 37:67–111, May 2000.

[11] A. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, Computer Laboratory, April 1996. Published as Technical Report No. 391.

[12] J. King, M. Lees, B. Logan. *Agent-based and continuum modeling of populations of cells*. Technical report, University of Nottingham, December 2006.

[13] E. A. Lee. *Overview of the Ptolemy project*. Technical memorandum UCB/ERLM01/11, Electronic Research Laboratory, University of California, Berkeley, March 2001.

[14] The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial version 1.4*, December 2000. http://www.modelica.org/documents/ModelicaTutorial14.pdf.

[15] The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification version 2.2*, February 2005. http://www.modelica.org/documents/ModelicaSpec22.pdf.

[16] P. J. Mosterman. *An overview of hybrid simulation phenomena and their support by simulation packages*. In F. W. Vaadrager and J. H. van Schuppen, eds., Hybrid Systems: Computation and Control '99, number 1569 in Lecture Notes in Computer Science, pages 165–177, 1999.

[17] P. J. Mosterman, G. Biswas, M. Otter. *Simulation of discontinuities in physical system models based on conservation principles*. In Proceedings of SCS Summer Conference 1998, pages 320–325, July 1998.

[18] H. Nilsson, A. Courtney, J. Peterson. *Functional reactive programming, continued*. In Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02), pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[19] H. Nilsson, J. Peterson, P. Hudak. *Functional hybrid modeling*. In Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages, volume 2562 of Lecture Notes in Computer Science, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.

[20] C. Nytsch-Geusen et al. *MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics*. In Proceedings of the 4th International Modelica Conference, Hamburg, Germany, March 2005. Modelica Association.

[21] C. C. Pantelides. *The consistent initialization of differential-algebraic systems*. SIAM Journal on Scientific and Statistical Computing, 9(2):213–231, March 1988.

[22] R. Paterson. *A new notation for arrows*. In Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming, pages 229–240, Firenze, Italy, September 2001.

[23] J. Peterson, G. Hager, P. Hudak. *A language for declarative robotic˙programming*. In Proceedings of IEEE Conference on Robotics and Automation, May 1999.

[24] J. Peterson, P. Hudak, A. Reid, and G. Hager. FVision: *A declarative language for visual tracking*. In Proceedings of PADL'01: 3rd International Workshop on Practical Aspects of Declarative Languages, pages 304–321, January 2001.

[25] F. Smith, D. Grossman, G. Morrisett, L. Hornof, T. Jim. *Compiling for run-time code generation*. Submitted for publication to JFP SAIG.

[26] Z. Wan and P. Hudak. *Functional reactive programming from first principles*. In Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation, pages 242–252, June 2000.

[27] Hongwei Xi and F. Pfenning. *Dependent types in practical programming*. In Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages, pages 214–227, San Antonio, January 1999.

[28] D. Zimmer. *Enhancing Modelica towards variable structure systems*. In Proceedings of 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT 2007), Berlin, Germany, July 2007. LiU E-Press.

Corresponding author: Henrik Nisson
School of Computer Science and IT
University of Nottingham, United Kingdom
nhn@cs.nott.ac.uk

Accepted EOOLT 2007, June 2007
Received: August 10, 2007
Accepted: September 10, 2007

TN

Structure of Simulators for Hybrid Systems − Development and New Concept of External and Internal State Events

Felix Breitenecker[1],. Günther Zauner[2], Nikolas Popper[2], Florian Judex[1], Inge Troch[1]
[1]Vienna Univ. of Technology, Austria, [2]dieDrahtwarenhandlung Simulation Services, Austria

At first this paper discusses discrete elements in the CSSL Standard, and- more detailed- the classification of 'classical' state events, where, structural-dynamic systems are generated by state events, changing the dimension of the state space. The paper continues with recent developments coming from Modelica and VHDL-AMS, which introduce non-causal modelling on a high level, including implicit models and state events associated with boundary conditions. While both new standards extend the CSSL standard, with focus on continuous systems; especially Modelica also allows defining pure discrete model constructs based on events, state charts, and Petri nets.

The main chapters concentrate on further extensions of the CSSL frames, mainly in order to handle hybrid and structural-dynamic systems properly. There, features of two competiting 'ideas' are sketched, maximal state space versus hybrid decomposition. In order to allow a highly flexible modelling level, state events are characterised as 'internal state events' (I-SE) or 'external state event' (E-SE). Both types of event can be described by state charts; implementation is the simulator's task. Finally, simulators being able to implement both state event types are reviewed: Modelica/Dymola, Mosilab, AnyLogic, and MATLAB/Simulink /Stateflow.

SNE 17/2, September 2007

Introduction

Since early times of simulation, attempts were made to standardize digital simulation programs by means of a self standing structure for simulation systems.

But only in 1968 the CSSL Standard (The CSSL Report commissioned by the Simulation Council Inc - SCI) became a milestone in the development: it unified the concepts and language structures of the available simulation programs, it defined a structure for the model, and it described minimal features for a runtime environment. In principle, this basic CSSL structure standard has been a standard for almost four decades, although a lot of extensions and other concepts have been developed and discussed. Also modern simulation systems, like Dimple, follow an extended CSSL standard. Mainly these extensions deal with discrete model parts and with DAE modelling.

An alternative standardised structure on basis of system theory is Zeigler's hybrid extension of the DES formalism (Discrete Event Systems). Unfortunately this approach is not commonly used in the area of continuous system modelling, and prototype implementations rather focus on the discrete world. Since three years a new idea is discussed in combined continuous / discrete modelling and simulation, namely, the challenge of structural dynamic systems.

Those can be modelled and simulated in CSSL structures, but there happen difficulties with the fixed state space, as well in the CSSL structure as in hybrid DES.

In any case, state events are bridging the gap between the continuous world and the discrete world. On the one side, continuous modelling considers state events to be an interruption of the continuous course of the system, which has to be handled properly, in order to continue continuously. On the other side, discrete system theory puts state events in the foreground, which update states and which control switching between different update algorithms (static algorithmic update, stochastic update based on event mechanisms, or also ODEs and DAEs).

Coming from the continuous side of simulation, state events may be viewed in an ambivalent way. They may cause discontinuous changes within a running algorithm updating the states (ODE solver), or they may cause a termination of the current update algorithm (ODE solver), starting a new update algorithm (ODE solver) with the same or with another model. Latest would allow easy modelling and simulation of structural dynamic systems.

Thus, it is worth to develop the mentioned idea based on a structure with *internal* and *external state events*, which will be discussed in chapter 5.

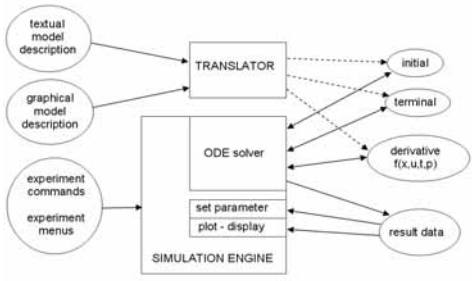


Figure 1. Basic Structure of a Simulation Language due to CSSL Standard

Interestingly, this idea of a distinction of state events is related to the development of Modelica, where *external state events* are discussed as basis for model switching based on state charts. Furthermore, since about five years it is tried to implement features for dynamic structures, by switching state events, in simulation systems. First, this was tried by means of simulator coupling, while now generic extensions and new systems are available (discussed in chapter 6).

1 CSSL standard

The CSSL standard suggests structures and features for a model frame and for an experimental frame. This distinction is based on Zeigler's concept of a strict separation of these two frames. Model frame and experimental frame are the user interfaces for the heart of the simulation system, for the simulator kernel or simulation engine. The simulation engine drives the calculations in the time domain. This basic structure of a simulator - due to CSSL standard – is illustrated in Figure 1.

There are not included any features for discontinuous changes, thus, very soon at least time event features were incorporated.

2 Discrete elements and events in continous simulation

The CSSL standard defines segments for discrete actions, which were at first mainly used for modelling discrete control. So-called DISCRETE regions or sections manage the communication to and from the continuous world and compute the discrete model parts.

These discrete section models discrete events, scheduled by time-dependent inputs (time events), or scheduled by state-dependent threshold functions (state events).

2.1 Time Events

In the graphical model description discrete controllers and the time delay could be modelled by a z-transfer block. If a discrete action is more complex, graphical descriptions have problems. For this purpose SIMULINK offers triggered submodels, which can be executed only at one time instant, controlled by a logical trigger signal. New versions of MATLAB also integrate a state machine (State Flow) for event control. Recently (2006) event control is supported in MATLAB/ Simulink by the *SimEvent Blockset*, offering also the entity concept.

In any case, the simulation engine has to handle an event list, representing the time instants of discrete action and the calculations associated with the action, where in-between consecutive actions the ODE solver have to be called.

2.2 State Events

Much more complicated, but defined in CSSL, are the so-called state events. Here, a discrete action takes place at a time instant, which is not known in advance, it is only known as a function of the states, described by a threshold function. This discrete action ('timeless' action) may simply change an input – or the structure of a system.

As example we consider the pendulum with constraints. If the pendulum is swinging, it may hit a pin positioned at angle φ_p with distance l_p from the point of suspension. After hit case the pendulum swings on with the position of the pin as the point of rotation and the shortened length $l_s = l - l_p$ and the angular velocity $d\varphi / dt$ is multiplied at position φ_p by l / l_s, etc.These discontinuous changes are state events. For state events the classical state space description is extended by the state event function $h(x)$, the zero of which determines the event:

$$\dot{\vec{x}}(t) = \vec{f}(\vec{x}(t), \vec{u}(t), t, \vec{p}), \quad h(\vec{x}(t), \vec{u}(t), t, \vec{p}) = \vec{x}_0$$

$$\dot{\varphi}_1 = \varphi_2,\ \dot{\varphi}_2 = -\frac{g}{l}\sin\varphi_1 - \frac{d}{m}\varphi_2,\ h(\varphi_1, \varphi_2) = \varphi_1 - \varphi_p = 0$$

The example involves two different events: change of parameter (length), and change of state (angular velocity). Generally, state events can be classified in four types:

1. parameter change: SE-P
2. one or more inputs change discontinuously: SE-I
3. one or more states change discontinuously: SE-S
4. the dimension of the state vector changes discontinuously: SE-D

State Events Type 1 (SE-P) could also be formulated by means of IF-THEN-ELSE constructs and by switches in graphical model descriptions, without synchronisation with the ODE solver. Big changes in parameters may cause problems for ODE solvers with step-size control.

State Events Type 2 (SE-I) are no real state events, they are time events – and listed here due to historic reasons.

State Events Type 3 (SE-S) are essential state events. They have to be located, transformed into a time event, and modelled in discrete model parts. In principle, these types of state events cannot exist, because a state variable cannot jump; jumps in states are caused by simplified modelling approaches.

In case of the pendulum, in reality the hit at the pin is not an event changing the velocity; it is a short physical process different to the oscillation process. The whole process may be seen as sequence of different processes: oscillation of long pendulum (differential equations) – hit at pin (event or differential equation) – oscillation of short pendulum (differential equations), etc.

State Events of Type 4 (SE-D) are essential ones and indicate a structural change in the model. In mechanical systems, they indicate a change of degrees of freedom.

Very often the threshold function switches between different algebraic constraints, so that these state events are coupled with differential-algebraic equations. In principle, these events may occur frequently, so that the system is called structural dynamic, because the dimension of the systems changes quasi-dynamically.

Two philosophies are found in handling these structural dynamic problems: a hybrid decomposition of the process, or making use of frozen states (combined with index reduction algorithms).

2.3 Handling of State Events

The handling of a state event requires four steps:

1. Detection: usually by checking the change of the signum-function of $h(x)$.
2. Localisation: algorithms make use of either iterative techniques, or of interpolation techniques for determining the time instant of the event with sufficient accuracy.
3. Handling: calculating / setting new parameters, inputs and states; switching to new equations.
4. Restart of the ODE solver (in a 'maximal' state vector), or starting another model (hybrid decomposition).

State events face simulators with severe problems. Up to now the simulation engine had to call independent algorithms, now a root finder for the state event function $h(x)$ needs results from the ODE solver, and the ODE solver calls the root finder by checking the sign of h.

Figure 2, an extension of Figure 1, shows the new more complex structure of calls between model frame, experimental frame, simulation engine and libraries. Basically, the kernel of the simulation engine has become an event handler. Furthermore it has to be noted, that not only classical time domain analysis by ODE solvers is offered, but also linear analysis by means of eigenvalue algorithms. Figure 2 also shows an interesting relation to discrete simulation: an event list manager has to be implemented, which can handle also pure discrete systems without any ODE.

In case of a structural change of the system equations (SE-D), simulators usually can manage only fixed structures of the state space.

In textual model description the DISCRETE construct allows to define events of any type, in graphical model descriptions calculations at discrete time instants are difficult to formulate within the continuous input/output form.

2.4 Classical implementations of the *Constrained Pendulum* model

In this example state events of type 1 (SE-P: discontinuous change of pendulum length) and type 3 (SE-S: change of angular velocity) are involved. Listing 1 presents parts of a classical ACSL model description, working with two discrete sections `hit` and `leave`, representing the two different modes. These sections model the events, which are scheduled by the `SCHEDULE` statement in the dynamic model description.

In pure graphical model descriptions we are faced-with the problem that calculations at discrete time instants are difficult to formulate. For the detection of the event SIMULINK provides the Hit Crossing block (Figure 2). This block starts state event detection (interpolation method) depending on the input,

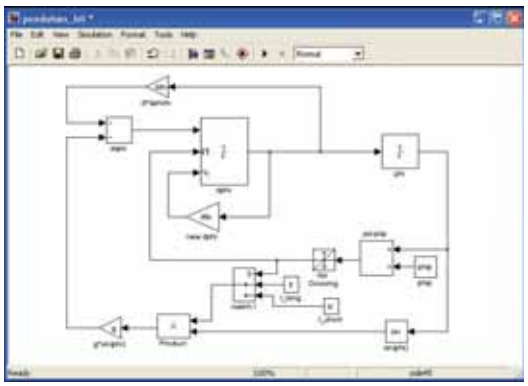


Figure 2. SIMULINK model of Constrained Pendulum

the state event function, and outputs a trigger signal.

For restarting the integration with new values for the angular velocity, a formulation going back to the times of analog computation is used: the integrator block is extended by a logical reset signal input; as this signal triggers, the integration is restarted with initial values fed into the initial value input. In this implementation, the new angular velocity is calculated continuously, while needed only at the hit event.

A more event-oriented implementation would make use of a triggered subsystem, which is executed only when the trigger from the hit crossing activates the event.

```
1  PROGRAM constrained pendulum
2     CONSTANT m = 1.02, g = 9.81, d =0.2
3     CONSTANT lf=1, lp=0.7
4     DERIVATIVE dynamics
5        ddphi = -g*sin(phi)/l - d*dphi/m
6        dphi = integ ( ddphi, dphi0)
7        phi = integ ( dphi, phi0)
8        SCHEDULE hit .XN. (phi-phip)
9        SCHEDULE leave .XP. (phi-phip)
10    END ! of dynamics
11    DISCRETE hit
12       l = ls; dphi = dphi*lf/ls
13    END ! of hit
14    DISCRETE leave
15       l = lf; dphi = dphi*ls/lf
16    END  ! of leave
17 END ! of constrained pendulum
```

Listing 1. ACSL textual model description

3 From CSSL to physical object-oriented modelling and state charts

In the 1990s, a lot of attempts have been made to improve and to extend the CSSL structure. The basic problem was the state space description, which limited the construction of modular and flexible modelling libraries. Two developments helped to overcome this problem. On modelling level, the idea of physical modelling gave new input, and on implementation level the object oriented view helped to leave the constraints of input/output relations. Furthermore, UML offers new input for hybrid modelling.

3.1 Physical modelling in Modelica and VHDL-AMS

A typical procedure for physical modelling is to cut a system into subsystems and to account for the behaviour at the interfaces. Each subsystem is modelled by balances of mass, energy and momentum and material equations. The complete model is obtained by combining the descriptions of the subsystems and the interfaces. This approach requires a modelling paradigm different to classical input/output modelling. A model is considered as a constraint between system variables, which leads naturally to DAE descriptions. The approach is very convenient for building reusable model libraries.

An international effort was initiated in September 1996 for the purpose of bringing together expertise in object-oriented physical modelling (portbased modelling) and defining a modern uniform modelling language, called Modelica. Modelica is intended for modelling within many application domains and their combinations. It supports several modelling formalisms: ODEs, DAEs, bond graphs, finite state automata, Petri Nets, etc. Modelica is intended to serve as a standard format so that models arising in different domains can be exchanged between tools and users.

Modelica is no simulator, Modelica is a modelling language, supporting and generating mathematical models in physical domains.

At the time the development of Modelica started, also a competitive development, the extension of VHDL towards VHDL-AMS was initiated. Both modelling languages aimed for general purpose use, but VHDL-AMS mainly addresses circuit design, and Modelica covers the broader area of physical modelling; modelling constructs such as Petri nets and finite automata could broaden the application area.

Modelica offers a graphical model frame, where the connections are bidirectional physical couplings. An example demonstrates how drive trains are handled. The drive train consists of four inertias and three clutches, where the clutches are controlled by input signals (Figure 3).

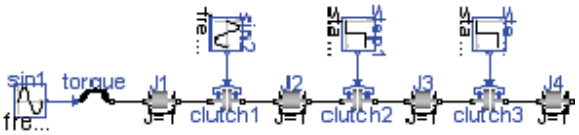


Figure 3: Graphical Modelica model for coupled clutches

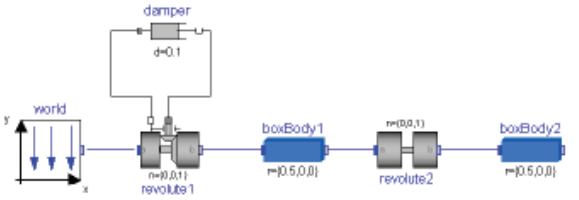


Figure 4: Graphical Modelica model for double pendulum

The graphical model layout corresponds with a textual model representation. This code can be changed and extended by the user, so that graphical and textual modelling can be combined. For example Figure 4 shows a graphical model of a double pendulum, consisting of two revolute joints (one with damper), and two masses modelling the rods. For joints and masses equations are predefined and sorted together during compilation.

Modelica can handle very different modelling approaches, not only ODEs and DAEs, but also finite state automata, and Petri nets. By means of state automata or state charts, conditions can be described more clear and transparent.

The translator from Modelica into the target simulator is not only able to sort equations; it also has to be able to process the implicit equations symbolically and to perform DAE index reduction.

Up to now, similar to VHDL-AMS, two simulation systems understand Modelica, Dymola from Dynasim, and MathModelica from MathCore Engineering. At present (2006/2007) the University of Lyngby develops and provides a Modelica simulation environment, the *Open Modelica* System, and Fraunhofer Gesellschaft develops a generic simulator, *Mosilab*, which understands Modelica models and supports variable dynamic structures.

The model for the constrained pendulum can be formulated in Modelica textually as a physical law for angular acceleration. The event with parameter change is put into an `algorithm` section, defining and scheduling the parameter event SE-P (Listing 2). Instead of angular velocity, the tangential velocity is used as state variable; the second state event SE-S 'vanishes'. In principle, one could use also graphical modelling for joint and mass using elements as in Figure 4, but the change of length has to be formulated textually in an `algorithm` section.

```
1 equation /*pendulum*/
2   v = length*der(phi); vdot = der(v);
3   m*vdot/length + m*g*sin(phi)+damp*v=0;
4 algorithm
5   if (phi<=phipin) then length:=ls; end if;
6   if (phi>phipin)  then length:=l1; end if;
```

Listing 2. Modelica model of Constrained Pendulum

3.2 Modelling events by state charts in AnyLogic

In the end of the 1990s, computer science put the simulator development forward. The Unified Modelling Language (UML) is one of the most important standards for specification and design of object oriented systems. This standard was tuned for real time applications in the form of a new proposal, UML for Real-Time (UML-RT). By means of UML-RT objects can hold the dynamic behaviour of an ODE. There exist a lot of simulation libraries for discrete simulation, based on the UML (class diagrams, state charts, etc). They allow for convenient modelling and simulation of *Discrete Event Systems (DEVS)*.

In 1999, a simulation research group at the Technical University of St. Petersburg used this approach in combination with a hybrid state machine for the development of a hybrid simulator, from 2000 on available commercially as simulator *AnyLogic*. The main building block is the *active object*. Active objects have internal structure and behaviour, and allow encapsulating of other objects to any desired depth. Relationships between active objects set up the hybrid model.

Active objects interact with their surroundings solely through boundary objects: ports for discrete communication, and variables for continuous communica-

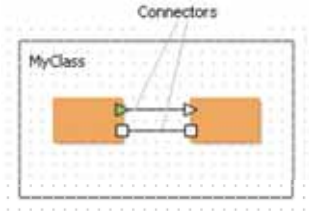


Figure 5. Acitive objects with connectors exchanging discrete messages (rectangle) and continuous signals (triangle)

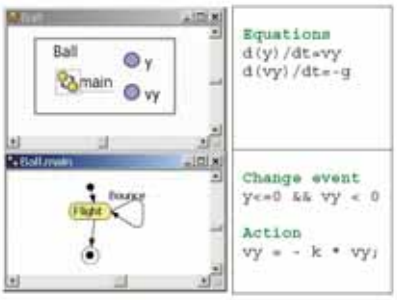


Figure 6. AnyLogic model for the *Bouncing Ball example; graphical modeling combined with the equation layer*

tion. The activities within an object are usually defined by statecharts. While discrete model parts are described by means of statecharts, events, timers and messages, the continuous model parts are described by ODEs and DAEs in CSSL-txpe notation and state charts.

The following AnyLogic implementation of the *Bouncing Ball* example shows a simple use of statechart modelling (Figure 6). The equations are defined in the active object ball, together with the state chart ball.main. This state chart describes the interruption of the state flight by the event bounce (SE-P and SE-S event).

4 Hybrid and structural-dynamic systems

Continuous simulation and discrete simulation have different roots, but they are using the same method, the analysis in the time domain.

In continuous and hybrid simulation the explicit or implicit state space description is used as common denominator. This state space may be described textually, by signal-oriented graphic blocks, or by power-based block descriptions. In discrete simulation we meet very different techniques for the model frame.

Application-oriented flow diagrams, network diagrams, state diagrams, etc. allow describing complex behaviour of event-driven dynamics. These descriptions are mapped to an event-based description.

On the other side, the simulator kernel is similar for discrete and continuous simulators. The model description is mapped to an event list with adequate update functions of the states within state update events. In discrete simulation the states are usually the status variables of servers and queues in the model, and state update is simple increase or decrease by increments.

In continuous simulation the state space is based on various laws used in the application area, and usually defined by DAEs. DAE solvers generate a grid for the approximation of the solutions. This grid drives an event list with state update events using complex formula depending on the chosen solver and on the defined DAE. Additional time events and state events are inserted into the global event list.

Hybrid systems often come together with a change of the dimension of the state space, then called *structural-dynamic systems*. The dynamic change of the state space is caused by a state event of type SE-D. In contrary to state events SE-P and SE-S, states and derivatives may change continuously and differentiable in case of structure change.

In principle, structural-dynamic systems can be seen from two extreme viewpoints. The one says, in a maximal state space state events switch on and off algebraic conditions which freeze certain states for a certain period. The other one says that a global discrete state space controls local models with fixed state spaces, whereby the local models may be also discrete or static. These viewpoints derive two different approaches for structural-dynamic systems, the *maximal state space*, and the *hybrid decomposition*.

4.1 Maximal State Space for structural-dynamic systems – internal events

Most implementations of physically-based models support a big monolithic model description, derived from laws, ODEs, DAEs, state event functions and *internal events*. The state space is maximal and static, index reduction in combination with constraints keep a consistent state space. Dymola, OpenModelica, and VHDL-AMS follow this approach.

This approach can be classified with respect to event implementation.It handles all events of any kind (SE-P, SE-S, and SE-D) within the ODE solver frame, also events which change the state space dimension (change of degree of freedoms) – consequently called *internal events*.

Using the classical state chart notation, *internal state events* **I-SE** caused by the model schedule the model itself, with usually different re-initialisations (depending on event type I-SE-P, I-SES, I-SE-D; Figure 7).

Modelica, VHDL-AMS, and Dymola follow this approach, handling also DAE models with index

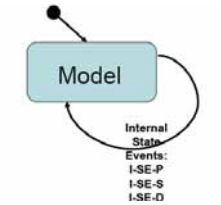


Figure 7: State chart control for *internal events* of one model

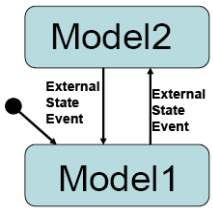


Figure 8: State Chart Control for External Events for two Models

higher than 1; discrete model parts are only supported at event level. MATLAB/Simulink also generates a maximal state space.

4.2 Hybrid Decomposition for structural-dynamic systems – external events

The hybrid decomposition approach makes use of *external events* (*E-SE*), which controls the sequence and the serial coupling of one or more models. A convenient tool for switching between models is a state chart, driven by *external events* – which itself are generated by the models. In the following example the UML-RT notation, control for continuous models and for discrete actions can be modelled by state charts. Figure 8 shows the hybrid coupling of two models, which may be extended to an arbitrary number of models, with possible events E-SE-P, E-SE-S, and E-SE-D. As special case, this technique may also be used for serial conditional 'execution' of one model –Figure 9 (only for SE-P and SE-S).

This approach additionally allows not only dynamically changing state spaces, but also different model types, like ODEs, linear ODEs (to be analyzed by linear theory), PDEs, etc. to be processed in serial or also in parallel, so that also co-simulation can be formulated based on external events. This approach allows handling all events also outside the ODE solver frame. After an event, a totally new model can be started. This makes sense especially in case of events of type SE-D and SE-S.

Figure 10 shows a structure of a simulator supporting this hybrid approach. Some work has to be investi-

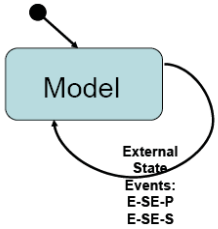


Figure 9: State Chart Control for External Events for one Model

gated into extension of e.g. Modelica for using this external control of models. The figure summarizes the outlined ideas by extending the CSSL structure by control model, external events and multiple models.

Clearly, not only ODE solver can make use of the model descriptions (derivatives), but also eigenvalue analysis, steady state calculation and other analysis algorithms may be used.

4.3 Mixed approach with internal/external events

A simulator structure as proposed in Figure 10 is a very general one, because it allows as well external and internal events, so that hybrid coupling with variable state models of any kind with internal and external events are possible (Figure 11).

Both approaches have advantages and disadvantages. The classical Dymola approach generates a fast simu-

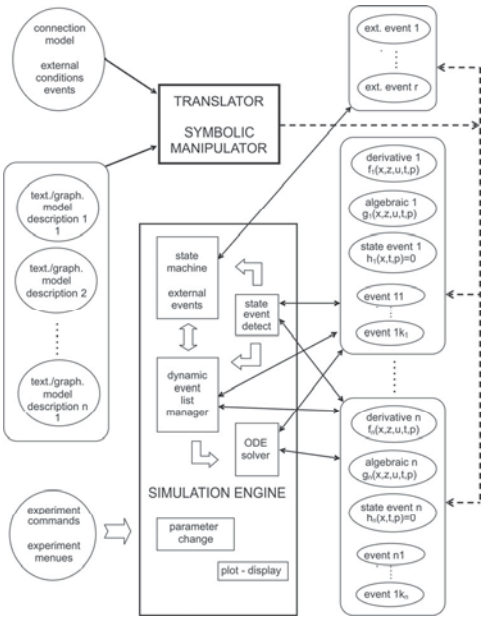


Figure 10: Structure for a simulation system with external state events E-SE and classical internal state events I-SE for controlling different models.

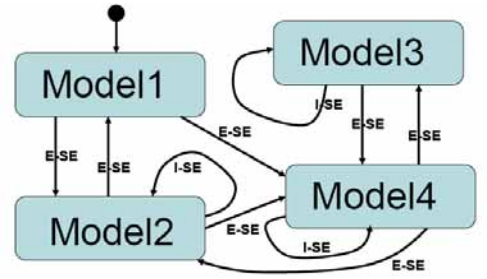


Figure 11. State chart control for different models with *internal* and *external* events

lation, because of the monolithic program. But the state space is static.

A hybrid approach handles separate model parts and has to control the external events. Consequently, two levels of programs have to be generated: dynamic models, and a control program – today's implementations are interpretative and not compiling, so that simulation time increases - but the overall state space is really dynamic.

- A challenge for the future lies in the combination of both approaches. The main ideas are:
- Moderate hybrid decomposition,
- External and internal events, and
- Efficient implementation of models and control.

For instance, for SE-P an implementation with an internal event may be sufficient (I-SE-P), for an event of SE-S type implementation with an external event may be advantageous because of easier state re-initialisation (E-SE-S), and for SE-D an implementation with an external event may be preferred (E-SE-D), because of much easier handling of the dynamic state change – and less necessity for index reduction.

An efficient control of the sequence of models can be made by state charts, but also by a definitions and distinction of *if*- and *when*- constructs, like discussed in extensions of SCILAB/SCICOS for Modelica.

5 Simulators for hybrid and structural dynamic systems

Up to now no simulator fulfils the structure given in Figure 12 completely. The main questions are:

- whether acausal physical modelling is supported,
- whether a-causal physical modelling is obeying the Modelica standard,

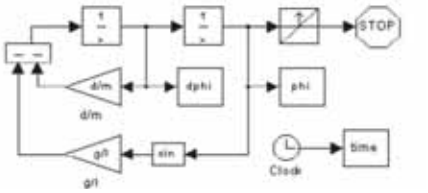


Figure 12: Simulink model for *Constrained Pendulum* with *external event* detected by hit-crossing block

- whether external events are supported (equal to whether hybrid decomposition into independent submodels is possible), and
- whether state chart modelling is supported.

In principle each combination of the above features is possible.

5.1 MATLAB/Simulink

The mainly interpretative system MATLAB/Simulink offers different approaches. First, it allows hybrid decomposition at MATLAB level. There, from MATLAB different Simulink models are called conditionally, and in Simulink a state event is determined by the hit-crossing block (terminating the simulation). For control, in MATLAB only if-then-else constructs and while structures are available (Listing 3, Fig. 12).

```
1  if ((phi_p-phi0)*phi_p<0 |
2           (phi0==phi_p & phi_p*v>0))
3     dphi0=v/ls;
4     sim('pendulum_short',[t(length(t)),
5     10]);
6     v=dphi(length(dphi))*ls;
7  else
8       dphi0=v/l;
9       ...
10 end
```

Listing 3. MATLAB control in *Constrained Pendulum* example for *external events* switching between long and short pendulum

At Simulink level, different submodels may be controlled by Stateflow, Simulink's state chart modelling tool. But the system generates in any case a maximal state space. In both cases, a-causal modelling is not supported. Currently a number of new toolboxes for physical modelling are under development or quite new on the market.

5.2 Dymola/Modelica

Modelica and Dymola have already been discussed in Section 4, together with examples also for the *Constrained Pendulum* example. Modelica clearly offers a-causal modeling, and so Dymola does.

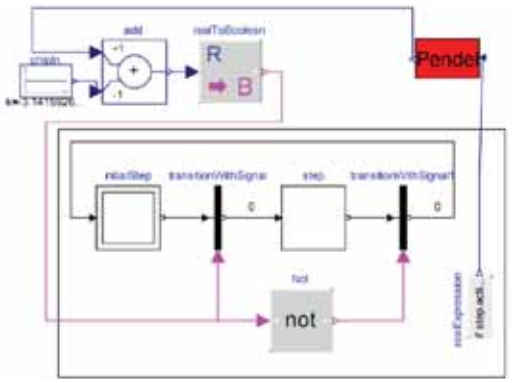


Figure 13: Graphical Dymola model for *Constrained Pendulum* with *internal events* managed by elements of Dymola's state chart library

But the Modelica definition says nothing about structural - dynamic systems, and Dymola builds up a maximal state space. Up to now there exit a Modelica standard library for state charts, but this construct is working only with internal events within the maximal state space. Figure 13 shows a *Constrained Pendulum* implementation with Dymola's state chart library.

5.3 Mosilab / Mosilab

At present Fraunhofer Gesellschaft Dresden develops in a cooperation of six institutes (FIRST, IIS/EAS, ISE, IBP, IWU and IPK) a generic simulator *Mosilab*, which defines an extension of Modelica: multiple models controlled by state automata. This simulator meets most of the challenges for the hybrid decomposition approach: at state chart level, state events of type SE-D control the switching between different models and service the events (E-SE-D). State events affecting a state variable (SE-S type) can be modelled at this external level (E-SE-S type), or also as classic internal event (I-SE-S). Also parameter events may be handled in both manners.

As first example, a model is presented, which describes the simplified dynamics of a landing device, which is falling and slowing down alternatively. The state chart in Figure 14 is translated into extended textual Modelica model description given in Table 4.

```
1  model System
2  statechart
3  state SystemSC extends State;
4    state Moving extends State;
5      state SlowDown extends State;...
6      end SlowDown;
7      State falling,State start(isInitial=true);
8     ...
9      transition t2: falling->slowDown event sw
10        guard sw==1 action body.add(boost)
11      end transition;...
12   end Moving;
13   State stop, start(isInitial=true);
14   Moving moving;
15   entry action  // executed, if state
16     SystemSC activ
17     gr := new Gravity();
18     boost := new Boost(empty=false);
19   end entry;...
20 end SystemSC;
21 end System;
```

Listing 4: Textual state chart notation for dynamics of landing device, Modelica extension in Mosilab

The dynamic models for the different phases may be modelled textually in Modelica standard or using elements from a graphical Modelica library. Mosilab translates each model separately, and generates a main simulation program from the state chart, controlling the call of the precompiled models and passing data between the models.

Mosilab is in development phase, so it supports only a subset of Modelica, and it does not perform index reduction, so that a-causal modelling is supported only at a lower level.

In a standard Modelica approach, the *Constrained Pendulum* is defined in the MOSILAB equation layer as implicit law (it is not necessary to transform to an explicit state space); the state event, which appears every time when the rope of the pendulum 'hits' or 'leaves' the pin, is modelled in an algorithm section with if (or when) – conditions.

Mosilabs state chart approach models discrete elements by state charts, which may be used instead of if- or when- clauses, with much higher flexibility and readability in case of complex conditions. There, *Boolean* variables define the status of the system and are managed by the state chart.

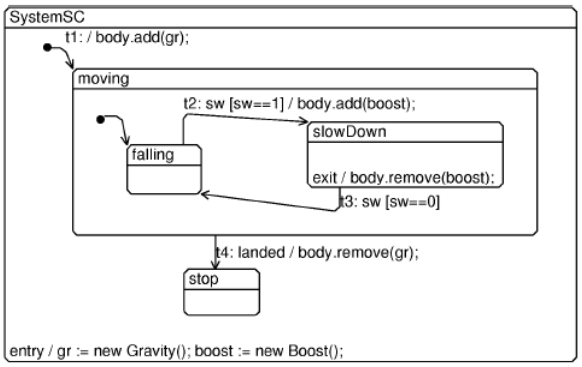


Figure 14: State chart for dynamics of landing device, Modelica Extension in Mosilab

The state charts initialize the system and manage switching between long and short pendulum, by changing the length appropriately (Listing 5).

```
1  event Boolean lengthen(start=false),
                 shorten(start=false);
2  equation
3    lengthen=(phi>phip); shorten=(phi<=phip);
4    v = l1*der(phi); vdot = der(v);
5    mass*vdot/l1+mass*g*sin(phi)+damping*v = 0;
6  statechart
7  state LengthSwitch extends State;
8    State Short,Long,Initial(isInitial=true);...
9    transition Long -> Short event shorten
10     action length := ls;
11   end transition;...
12 end LengthSwitch;
```

Listing 5. Mosilab model for *Constrained Pendulum* – state chart model with *internal events* (I-SE-P)

From the modelling point of view, this description is equivalent to the description with if-clauses. The Mosilab translator clearly generates an implementation with different internal equations. Mosilabs simulator performs simulation by handling the state event within the integration over the simulation horizon.

Mosilabs state chart construct is not only a good alternative to if- or when- clauses within one model, it offers also the possibility to switch between structural different models. This very powerful feature allows any kind of hybrid composition of models with different state spaces and also of different type (example see listing 6).

```
1  model Long
2  equation
3    mass*vdot/l1+mass*g*sin(phi)+damping*v = 0;
4  end Long;// the same for model Short with other paramaters
5  event discrete Boolean lengthen(start=true),
                          shorten;
6  equation
7    lengthen=(phi>phipin);shorten=(phi<=phipin);
8  statechart
9  state ChangePendulum extends State;
10   State Short, Long,
           startState(isInitial=true);
11   transition startState -> Long action
12     L:=new Long(); K:=new Short(); add(L);
13   end transition;
14   transition Long->Short event shorten action
15     disconnect …; remove(L); add(K); connect …
16   end transition;
17 end ChangePendulum;
```

Listing 6. Mosilab model for *Constrained Pendulum* – state chart switching between different pendulum models by *external events* (E-SE-P)

In case of the constrained pendulum, the system is decomposed into two different models, *Short pendulum* model, and *Long pendulum* model, controlled by a state chart. The state chart creates first instances of both pendulum models during the initial state (new). The transitions organise the switching between the pendulums (*remove, add*).

5.4 AnyLogic

AnyLogic, already discussed in Section 4, is based on hybrid automata. Therefore hybrid decomposition and control by external events is possible. AnyLogic can deal partly with implicit systems, but does not support a-causal modelling. Furthermore, new versions of AnyLogic concentrate more on discrete modelling and modelling with *System Dynamics*, whereby state event detection has been sorted out. For the *Constrained Pendulum* example, a hybrid decomposed model may make use of a model structure 'similar' to that one in Figure 6, but now two sets of the state equations are found in the sub states *Short* and *Long*. The events defined at the arcs stop the actual model, set new initial conditions and start the alternative model.

References

[1] Strauss, J. C. 'The SCi continuous system simulation language (CSSL)', Simulation 9, 281-303. San Diego: SCS Publishing, 1967.

[2] P. Fritzson: Principles of Object-Oriented Modeling and Simulation with Modelica, Wiley IEEE Press, ISBN 0-471-471631, 2005.

[3] C. Nytsch-Geusen, P. Schwarz, 'MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics', In Proc. 4th Modelica Conference TU Hamburg-Harburg, pp 527 – 535, 2005;

Corresponding author: Felix Breitenecker
Vienna University of Technology
Department of Analysis and Scientific Computing,
Wiedner Hauptstraße 8-10, 1040 Vienna, Austria
Felix.Breitenecker@tuwien.ac.at

Accepted EUROSIM 2007, June 2007
Received revised contribution: August 24, 2007
Revised: September 2, 2007
Accepted: September 5, 2007

Modeling Structural - Dynamics Systems in MODELICA/Dymola, MODELICA/Mosilab and AnyLogic

Günther Zauner, Felix Breitenecker, Vienna University of Technology, Austria
Daniel Leitner, Austrian Research Centres, Austria

With the progress in modeling dynamic systems new extensions in model coupling are needed. The models in classical engineering are described by differential equations. Depending on the general condition of the system the description of the model and thereby the state space is altered. This change of system behavior can be implemented in different ways. In this work we focus on three state-of-the-art DAE simulation environments, Dymola, Mosilab and AnyLogic, and compare the possibilities of coupling of different state spaces. This can be done either using a parallel model setup, a serial model setup, or a combined model setup. The analogies and discrepancies are figured out on the basis of the classical constrained pendulum as defined in ARGESIM comparison C7.

SNE 17/2, September 2007

Introduction

In the last decade the increase of computer power and the apace growth of model complexity leads to a new generation of simulation environments. Concurrently ambitions pointed towards establishing standardization. Especially Modelica organization develops a range of syntax description and standard libraries.

This paper will compare the solutions of the constrained pendulum as an easy to model example, implemented in the most common Modelica simulator Dymola, Mosilab, a product from six Fraunhofer Institutes which uses Modelica syntax with extensions for state charts, and the simulator AnyLogic from Xjtek in St. Petersburg. This simulator also has object oriented structure and is implemented in Java.

We will focus on how the model can be implemented and we will have a look in which time slot the state events are and if there is a significant difference referring to the implementation method.

1 Model

The constrained pendulum is a classical nonlinear model in simulation techniques. This model has been presented in the definition of ARGESIM comparison C7 [1]. There is no exact analytical solution to this problem. Therefore, the results have to be obtained by numerical methods. In this section a description of the model will be given.

The motion of the pendulum is given by

$$ml\ddot{\varphi} = -mg\sin(\varphi) - dl\dot{\varphi}, \qquad (1)$$

where φ denotes the angle measured in counter clockwise direction from the vertical position. The parameter m is the mass and l is the length of the pendulum. Damping is realized with the constant d.

In the case of a constrained pendulum a pin is fixed at a certain position given by the angle φ_p and the length l_p. If the pendulum is swinging it may hit the pin. In this case the pendulum swings on with the position of the pin as the point of rotation and the shortened length $l_s = l - l_p$.

Two experiments have been defined. The first one is starting in the long pendulum modus and is swinging towards the pin. The second experiment is a model where the starting conditions are set in a way that the pendulum is shortened in the beginning of the simulation run.

2 Simulation environments

In this section the focus is on three simulation environments. Two simulators, namely Dymola and Mosilab, are based on the model description standard Modelica [2]. Modelica is a freely available, object-oriented language for modeling of large, complex, and heterogeneous physical systems.

One of its most important features is non-causal modeling. In this modeling paradigm, users do not specify the relationship between input and output signals directly, but they rather define variables and the equations that must be satisfied.

It is suited for multi-domain modeling and control subsystems and process oriented applications. Modelica is designed that it can be utilized in a similar way as an engineer builds a real system: first trying to find standard components like motors, pumps and valves

from manufacturers catalogues with appropriate specifications and interfaces and only if there does not exist a particular subsystem, a component model would be newly constructed based on standardized interfaces.

The actual version of the Modelica Standard Library is 2.2.1, which has been released in April 2006.

2.1 Dymola

Dymola, DYnamic MOdeling LAboratory, is an environment for modeling and simulation of integrated and complex systems. It has unique multi-engineering capabilities which mean that models can consist of components from many engineering domains.

The basic structure of the simulator is divided into two separate parts: the Modeling layer and the Simulation layer. Thereby the modeling layer is separated in three parts. One part, the so called ICON layer, is used to define the shape of the new defined blocks. The DIAGRAM layer is the interface for graphical modeling. The third plane is the MODELICA TEXT part where the Modelica source code can be implemented directly.

Dymola has a strong focus on using symbolic methods for mass-matrix inversion and equation sorting.

Integration algorithms for non-real-time simulation typically handle discontinuities by detecting when certain variables cross a boundary. They then calculate the time of the event by iteration and then change the step size to advance the time exactly to the time of the event (crossing) [3].

The default integration method is the Dassl code as defined by Petzold. The method can also be freely chosen out of 15 standard solvers, including algorithms for stiff systems. There is until now no possibility implemented to make graphical model switching for subsystems with different state space dimension.

2.2 Mosilab

The simulator Mosilab (MOdeling and SImulation LABoratory) is an environment developed from the Fraunhofer-Institutes FIRST, IIS/EAS, ISE, IBP, IWU and IPK in the research project GENSIM.

It has been developed for time-continuous and time-discrete analysis of heterogeneous technical systems. The main innovation from point of simulation techniques view in this simulator is the illustration of condition-based changes in the model structure (model structure dynamics). With this mechanism it is possible develop and simulate models with different modeling depth.

The model description in general is done in the Modelica standard. Additional features to assure high flexibility during modeling and the concept of structural dynamics is implemented. This is done by extending the Modelica standard with state charts, controlling dynamic models. The extended object-oriented model description language resulting is called MOSILA [1,4] Moreover simulator coupling with standard tools (e.g. MATLAB/Simulink, FEMLAB) is realized.

Code generation is done in a quite similar way as in Dymola/Modelica. This makes sense, because this relatively new simulator will also be able to simulate problems defined in the standard Modelica notation with other tools, which use the same syntax. The main difference is the extension for graphical representation of state charts. This is solved with an interface where the user can define UML statecharts.

The analysis part of the model is split into two layers: the simulation and the post processing layer. The defined code is translated into C++. The default integration method is the so called IDADASSL.

2.3 AnyLogic

AnyLogic is a multiparadigm simulator supporting Agent Based modeling as well as Discrete Event modeling, which is flowchart-based, and System Dynamics, which is a stock-and-flow kind of description. Due to its very high flexibility AnyLogic is capable of capturing arbitrary complex logic, intelligent behavior, spatial awareness and dynamically changing structures. It is possible to combine different modeling approaches making AnyLogic a hybrid simulator. AnyLogic is highly object oriented and based on the Java programming language

The development of AnyLogic in the last years has been towards business simulation. In version 6 of AnyLogic it is possible to calculate problems from engineering, but there are certain restrictions. For example the integration method cannot be chosen freely and there is no state event finder.

When a model starts, the equations are assembled into the main differential equation system. During the simulation, this DES is solved by one of the numerical methods built in AnyLogic. AnyLogic provides a set of numerical methods for solving ordinal differen-

tial equations (ODE), algebraic-differential equations (DAE), or algebraic equations (NAE).

AnyLogic chooses the numerical solver automatically at runtime in accordance to the behavior of the system. When solving ordinal differential equations, it starts integration with forth-order Runge-Kutta method with fixed step. Otherwise, AnyLogic plugs in another solver—Newton method. This method changes the integration step to achieve the given accuracy.

3 Solution methods

New advantages in computer numerics and the fast increase of computer capacity lead to necessity of new modeling and simulation techniques. In many cases of modern simulation problems state events have to be handled.

There exit more or less different categories of structural dynamic systems which should be focused on and solved.

The first class of hybrid systems are the one, where the state space dimension does not change during the whole simulation time and also the system equations stay the same. Only so called parameter events occur at discrete time points. These are the more or less simplest form of state events. Modern simulators offer different solution methods. A Part of them have a *discrete section* or as implemented in Dymola and Mosilab a so called *algorithm section*. In this part the user can define the parameter value change using the commands *when, if*, etc. In this section the use of a causal modeling has to be switched off. This means that we have to make assignments for the parameter values at time point the event occurs.

Furthermore many software environments support the usage of UML state charts. This is a very intuitive and convenient way to describe a system which contains multiple discrete states. In the combination with dynamical equations this approach enables a simple

Figure 2. The parameters of the model are changed by an UML state diagram.

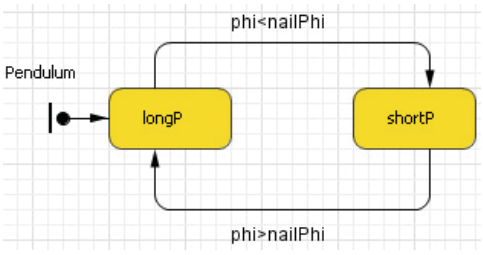


Figure 1. UML state diagram controlling the pendulum.

implementation of structural dynamics. The dynamic equations or parameters are dependent of the discrete state of the model. On the other hand the states can be altered in dependence of the dynamic variables.

In case of the constrained pendulum the states are normally swinging (state 'long') or swinging with shortened length around the pin (state 'short'). The discrete state of the model depends on the angle φ and the pins angle φ_p. The state alters the model parameters or the models set of equations, see figure 1.

3.1 Switching states

When the state of a system changes, often the state space of the model stays unchanged, thus the same set of differential equation can be used for different states. In this situation only certain parameters must be changed when a state is entered.

In case of the constrained pendulum the differential equation for movement stays the same for both states 'long' and 'short'. If the state changes the parameter length and angular velocity are updated before the calculation can continue, see figure 2.

When the state of a system changes, often the state space of the model stays unchanged, thus the same set of differential equation can be used for different states. In this situation only certain parameters must be changed when a state is entered.

In case of the constrained pendulum the differential equation for movement stays the same for both states 'long' and 'short'. If the state changes the parameter length and angular velocity are updated before the calculation can continue, see figure 2.

3.2 Switching models

Often the previous approach is not possible. Sometimes situation occur where the state space of the model changes, thus a simple change of parameters is not possible. Normally the whole set of differential equations, thus the complete model, must be changed.

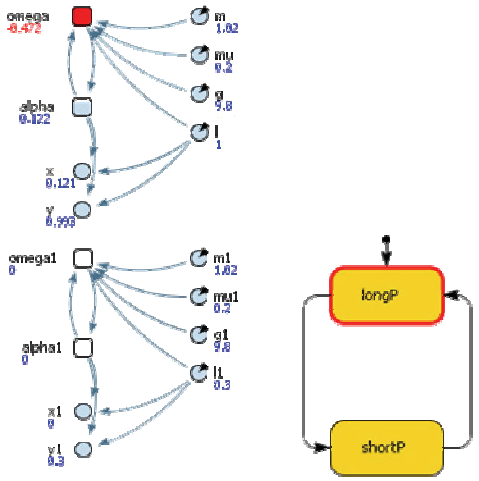


Figure 3. The differential equations of the system are switched in dependence of the UML state diagram.

In many simulation environments this approach can lead to complication.

In case of the constrained pendulum two differential equations are set up describing the movement of the pendulum. One describes the normal pendulum the other one the shortened pendulum. Which equation is set to be active is determined by the state diagram. When the states are switched the initial values must be passed on the equation must be activated and the other one must be frozen, see figure 3.

4 Dymola

The implementation of the constrained pendulum has been done in two more or less different ways. As Dymola does not support the UML notation for state charts and there is in the moment no method implemented to switch between two or more independent models during one simulation run, the solution methods described in section 3.1 and 3.2 can not be used.

In our example the state event, which appears every time when the rope of the pendulum hits the pin or looses the connection to it, is modeled in an *algorithm* section. This can be done with the following code digest:

```
1 algorithm
2    if (phi<=phipin) then
3       length := ls;
4    end if;
5    if (phi>phipin) then
6       length := l1;
7    end if;
```

Another method for implementing the constrained pendulum in Dymola is the use of standard blocks in combination with a predefined model which includes the equations or using only the *Modelica.Blocks* components.

In this example the solution is made by using standard blocks with little extension. Figure 4 shows a screenshot of the Diagram layer of this model.

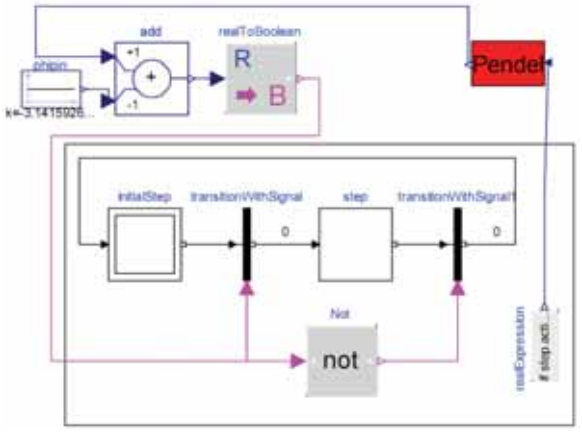


Figure 4. The screenshot of the Diagram layer in Dymola/Modelica.

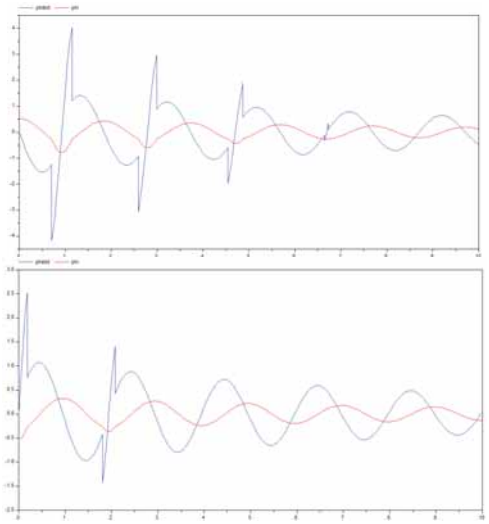
The simulations are done for both tasks and the solutions are compared. This is done by plotting all the results in one picture. The time of the last event in task a (figure 5) is in both cases the same, namely 6.72198 seconds. There is no easy possibility to plot the difference of special variables from different simulation runs. The same model has to be checked with other starting values. This is done in next step. The figure 6 shows the plot for starting angle $\varphi = -\frac{\pi}{6}$ instead of $\frac{\pi}{6}$.

5 Mosilab

Similar to the way the solutions in Dymola were calculated, the system can be solved with Mosilab. But as mentioned before, this structure can not handle changes in the state space dimension. The implemented Modelica extension enables the handling of discrete elements as well as structure changes in the general description.

We focus on two different solution methods for the constrained pendulum.

First approach: State charts may be used instead of `if`- or `when`- clauses (similar to 3.1 Switching states), with much higher flexibility and readability in case of complex conditions. Boolean variables define the status of the system and are managed by the state



Figures 5, 6. Angle (red, inner graph) and angular velocity (blue) as described in section 1.

chart. The most important part of the source code is as follows:

```
1  equation
2     lengthen = (phi > phipin);
      shorten = (phi <= phipin);
3     /* pendulum equations here */
4  statechart
5  state LengthSwitch extends State;
6     State Short, Long, Initial(isInitial=true);
7     transition Initial -> Long end transition;
8     transition Long -> Short event shorten
         action length := ls;
      end transition;
9     transition Short -> Long event lengthen
         action length := l1;
      end transition;
10 end LengthSwitch;
```

From the modeling and mathematical point of view, this description is equivalent to the description with `if`-clauses. The question is, how the Mosilab translator generates the implementation of the equations in both cases. The Mosilab/Modelica simulator performs simulation by handling the state event within the integration over the simulation horizon.

Second approach: These models are the conversion of concepts from section 4.2, which is switching models into Mosilab notation. For the constrained pendulum, we decompose the system into two different models, a `short` and a `long` pendulum model, controlled by a state chart. This can again be done with graphical aid in the form of UML diagrams.

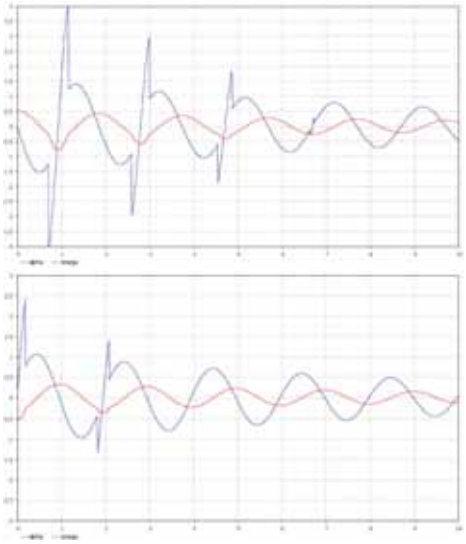
In the development status at the end of 2006, there still occurred several problems with the graphical interface of the state chart layer. The functionality of the system is not restricted. The results are similar to the solutions done with Dymola/Modelica.

6 AnyLogic

The implementation of the constrained pendulum has been done in two different ways. In the first approach only the parameter states have been switched corresponding to section 4.1, in the second approach the whole differential equation is switched corresponding to section 4.2. Both examples from chapter 2 have been calculated with both approaches. The results in AnyLogic are identical in both methods because the times of the state transitions are the same.

In the first approach the model consists of two ordinary differential equations describing the movement of the pendulum. In these equations four parameters are used length l, mass m, damping d, and gravity g. Further a state diagram with states *long* and 'short' and two transitions are used to update the equations. When the state changes length l and angular velocity ω are updated. The results calculated by AnyLogic 6 are plotted in figures 7 and 8.

The second approach uses two separate models. The implemented model consists of two times two ordinary differential equations. Both equations have four parameters separately: length l, mass m, damping d, and gravity g. A state diagram is implemented analog to the first approach. If the state changes the right differential equations are activated and their



Figures 7, 8. Results for example 1 and 2, respectively: angle (red, inner graph), angular velocity (blue).

initial values are set, while the other differential equation is frozen.

7 Discussion

For this nonlinear model, there exists no exact solution. For this reason we can only calculate the numerical solutions and compare, for example, the time points where the last state event appears. This is the moment when the rope of the pendulum looses the connection to the pin the last time. In the first model under investigation, this happens after the fourth time shortening the pendulum, which means after eight state events all together. In the second simulation run, this occurs earlier, namely already after two times lengthening the rope, which means after three state events, because of the special initial condition (pendulum is in short modus at starting time).

The solutions are calculated with the default simulation method, if possible. With this approach we try to test the simulation environments from the user's point of view. Many programmers and modelers do not care that much about the implemented integration methods. For this reason the standard method has to produce reliable results in an appropriate calculation time.

The solution in the Mosilab simulator with standard Modelica components cannot be calculated with the standard method (*Dassl* code), because during simulation of this task a numerical error occurs and therefore the calculation is interrupted. The integration method pins at the time point of the first state event. Because of this reason the *Implicit Trapez* method was chosen. The other results are all done with the standard integration method and the given step sizes/number of intervals.

Simulator	Simulation	Method
Dymola/Modelica	6.72198	Dassl 500 intervals
Mosilab/Modelica Switch models	6.7204	IDA Dassl Min. step 1e-6 Max. step 0.08
Mosilab/Modelica Pure Modelica	6.7199	Impl. Trapez Min. step 1e-6 Max. step 1e-4
Mosilab/Modelica Parameter switching	6.7224	IDA Dassl Min. step 1e-6 Max. step 0.08
AnyLogic	6.725	No influence Step size 0.001

Table 1. End time of the last shortening of the pendulum for example 1.

Table 1 shows that the solutions with Dymola and Mosilab are equivalent, if the solution is rounded towards two digits after the comma. By contrast, the solution in AnyLogic differs. We can try to explain this difference by taking a look on state event finding. This is not implemented in AnyLogic and is missing as an important standard feature of modern simulation environments. The lack of influence on the numerical methods can be explained by the main field of application of AnyLogic. Its main focus is on production and logistics, not on simulation of DAE systems.

In table 1 we see that there is only one row for Dymola/Modelica. This is because of equivalent results in all three implementations. Also AnyLogic delivers the same result for both methods. As we see, in this case Dymola outperforms Mosilab, because the result does not depend on the way of implementation. On the other hand we cannot implement real structural dynamics without blowing up the state space and problems in starting variable definition.

The graphical user interface for UML diagrams is a big advantage of Mosilab and AnyLogic compared to the possibilities of Dymola. But we have to keep in mind, that this feature is not Modelica standard, which complicates model exchange between different simulators based on Modelica.

References

[1] Nytsch-Geusen, C. et. al. Advanced modeling and simulation techniques in MOSILAB: A system development case study. Proc. of the 5th International Modelica Conference, 2006.

[2] P. Fritzson, 2004. Principles of Object Oriented Modeling and Simulation with Modelica 2.1., IEEE Press, John Wiley&Sons, Inc., Publication, USA.

[3] H. Elmquist, et. al. Real-time Simulation of Detailed Automotive Models, Proceedings of the 3rd International Modelica Conference, Linköping, Sweden

[4] T. Ernst, A. Nordwig, C. Nytsch-Geusen, C. Claus, A. Schneider: MOSILA Modellbeschreibungssprache, Spezifikation, Version 2.0, from the homepage: www.mosilab.de/downloads/dokumentation

Corresponding author: Günther Zauner
Vienna University of Technology
Department of Analysis and Scientific Computing
Wiedner Hauptstraße 8-10, 1040 Vienna, Austria
gzauner@osiris.tuwien.ac.at

Accepted EOOLT 2007, June 2007
Received: September 10, 2007
Revised: September 20, 2007
Accepted: September 25, 2007

Proceedings CD der
Konferenz zur Multiphysik-Simulation

ANWENDUNGSBEREICHE:

- Akustik und Fluid-Struktur-Interaktion
- Brennstoffzellen
- Chemietechnologie und Biotechnologie
- COMSOL Multiphysics™ in der Lehre
- Elektromagnetische Wellen
- Geowissenschaften
- Grundlegende Analysen, Optimierung, numerische Methoden
- Halbleiter
- Mikrosystemtechnik
- Statische und quasi-statische Elektromagnetik
- Strömungssimulation
- Strukturmechanik
- Wärmetransport

Bestellen Sie hier Ihre kostenlose Proceedings CD mit Vorträgen, Präsentationen und Beispielmodellen zur Multiphysik-Simulation:

TITEL, NACHNAME

VORNAME

FIRMA / UNIVERSITÄT

ABTEILUNG

ADRESSE

PLZ, ORT

TELEFON, FAX

EMAIL

www.comsol.de

Fax: +49(0)551 / 99 721- 29
www.comsol.de/conference2005/cd/

Nachdem der Endabstieg der beiden Mars Rover unter Tausenden von atmosphärischen Bedingungen simuliert wurde, entwickelte und testete das Ingenieur-Team ein ausfallsicheres Bremsraketen-System, um eine zuverlässige Landung zu garantieren. Das Resultat – zwei erfolgreiche autonome Landungen, die exakt gemäß der Simulation erfolgten.
Mehr hierzu erfahren Sie unter:
www. mathworks.de/mbd

MATLAB®
&SIMULINK®

The MathWorks
Accelerating the pace of engineering and science