Introducing Explicit Causality in Object-oriented Hybrid System Modeling

Liu Liu, Felix Felgner, Georg Frey

9th International Conference of Modeling, Optimization and Simulation - MOSIM'12
June 06-08, 2012 – Bordeaux - France
"Performance, interoperability and safety for sustainable development"

INTRODUCING EXPLICIT CAUSALITY IN OBJECT-ORIENTED HYBRID SYSTEM MODELING

Liu Liu, Felix Felgner, Georg Frey

Chair of Automation, Saarland University
Saarbrücken, Germany
{ liu.liu | felix.felgner | georg.frey }@aut.uni-saarland.de

ABSTRACT: *Along with the rapid development of embedded devices and network technology, the area of Cyber Physical Systems (CPS), has arisen. In terms of modeling and simulation, CPS—like many technical systems—have a hybrid nature, i.e., discrete-event behavior and continuous-time dynamics have to be integrated with each other. Basically, this integration is supported by modern object-oriented modeling paradigms such as Modelica®. The equation-based concept resolves the causality between interconnected components, which qualifies this modeling scheme for complex multi-domain systems. However, in hybrid systems, explicit causality is required to correctly manage iterative events. This paper highlights these issues, including algorithmic loops and instantaneous multiple updates, which essentially arise from incompatibilities between the object-oriented concept and specific discrete-event phenomena. We discuss several possible solutions and introduce the concept of re-allocating the objects' behavioral intelligence.*

KEYWORDS: *Object-oriented Modeling and Simulation, Causality, Cyber-Physical System (CPS), algorithmic loop, instantaneous multiple updates*

1 INTRODUCTION

The modeling and simulation of hybrid systems has been discussed for more than 30 years, with F. E. CELLIER's 1979 dissertation (Cellier, 1979) being one of the earliest extensive contributions in the field. By now, various approaches have been developed to model and analyze these systems, e.g. using hybrid automata (Henzinger, 1996) in system modeling and model checking (Henzinger, 1997) for system analysis. Regarding simulation, hybrid simulation technology is basically divided into two fractions: Firstly, the well-known technique—applied, e.g., by Matlab® and Dymola®—making use of modern ODE solvers with additional mechanism for handling discontinuity. Secondly, a relatively new approach introduced in PowerDEVS (Bergero and Hofman, 2011), which applies a quantization-based integration method and the Discrete Event System Specification (DEVS) formalism (Zeigler et al., 2000) for the modeling and simulation of hybrid systems.

The term hybrid system is used to classify a system which involves combined continuous and discrete behavior. Plenty of examples can be categorized as hybrid systems, either pure physical systems (e.g. bouncing ball) or control systems (e.g. thermostat). Along with the rapid development of embedded devices and network technology, a more specific sub-area of hybrid systems, the Cyber Physical Systems (CPS), has arisen. CPS emphasizes the integration of computation and physical processes (Lee, 2008). The author in (Lee, 2008) points out some interesting challenges in the design process of CPS. Some of the challenges, e.g. handling concurrency, building more deterministic models and managing large scale heterogeneous structures, are also of interest from the simulation point of view. Consequently, the analysis of CPS requires practice- and engineering-oriented tools. The classic methods based on rigorous mathematical models such as hybrid automata and DEVS, due to the lack of commercial-level software supports, may no longer be efficient or even not applicable to treat CPS.

The object-oriented (OO) modeling paradigm, to be considered as the de-facto standard to handle the system complexity, provides a further potential framework for the modeling and simulation of hybrid systems, especially for CPS.

Recently, several works have been started in the field of modeling and simulation of CPS using the Modelica® language. The 'NCLib' library presented in (Liu and Frey, 2008), (Wagner et al., 2008) provides components for modeling computation and communication systems. Detailed timing effects, including real-time task scheduling, task execution and network communication, are captured. In combination with other Modelica® libraries, complex CPS can be modeled and simulated in a single simulation environment. An illustrative CPS example is given in Figure 1 (Liu and Frey, 2008). Here, the OO modeling paradigm is extensively employed to break down the complexity of the whole system which involves ca. 13,000 variables. The robotic arm consists of five revolute joints, each with a drive axis attached. An embedded controller (axis controller) is deployed for each drive axis. The axis controllers exchange infor-

mation (sensor values, actuator values) with the center controller (trajectory plan) through the Fully-Switched-Ethernet. The robotic arm is modeled using the Modelica® Multibody library, while the communication and computation components are built upon components from the NCLib library. Besides 'NCLib', 'TrueTime Network' (Reuterswärd et al., 2009) transplants the network part of the 'TrueTime' library from MATLAB/Simulink to Modelica®. The 'Modelica_EmbeddedSystems' library presented in (Henriksson and Elmqvist, 2011) shows recent Modelica® developments facilitating integrated model-based system development applicable to CPS.

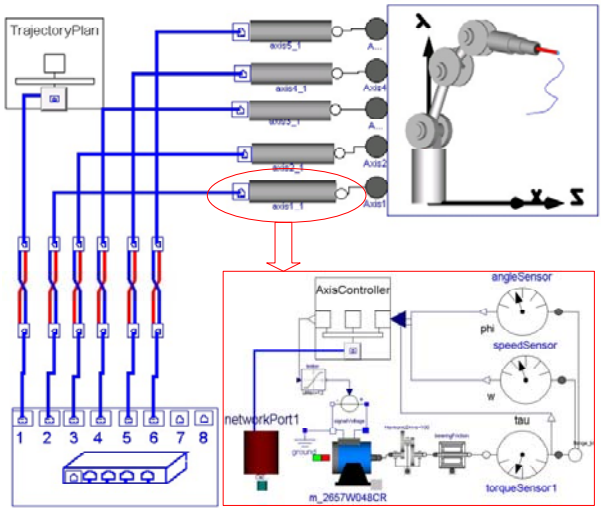

Figure 1. Modeling a CPS using 'NCLib'

One main challenge of hybrid system simulation is handling the synchronization and the temporal or logical ordering of multiple events, for which we use the term *causality*. This is especially true regarding the aforementioned CPS, as stated in one of E.A LEE's recent publications:

"It is also challenging to accurately represent in models distinct events that are causally related but occur at the same time." (Derler et al., 2011)

In order to achieve deterministic behavior in simulation, the causality of the modeled systems has to be modeled unambiguously and simulated correctly. This challenge also applies to the state-of-the-art OO modeling languages and simulation packages (e.g. Modelica® / Dymola®, which is further discussed in this paper).

This paper aims at proposing a well-arranged and systematical solution for the causality problem. Based on a detailed discussion of existing approaches, a new approach called *intelligence re-allocation* is suggested. This approach is based on a modified OO modeling paradigm. Since it utilizes as less tool-specific features as possible, it is a promising generic method for the OO modeling and simulation of hybrid systems regarding the causality problem. This approach has been utilized in the implementation of the 'NCLib' library.

The paper is structured as follows: Section 2 introduces the origin of the causality problem in OO modeling and simulation. The way of handling causality in modern simulation tool Modelica® / Dymola® and potential problems are shortly introduced. Section 3 explains the problem of *algorithmic loop* and discusses three possible solutions. In section 4, the *instantaneous multiple updates problem*, as a further complication of the causality problem, is introduced and three feasible solutions are analyzed. As a generic solution to resolve causality problems, the *re-allocation of objects' behavioral intelligence* is proposed in section 5. Section 6 concludes the paper.

2 THE CAUSALITY PROBLEM

2.1 The origin of the causality problem

One of the intrinsic characteristics of every real, physical system is *concurrency*. Interactive elements in a physical system also behave concurrently. During the modeling of such a system, certain interactions may be abstracted, typically those interactions which (from the system point of view) serve for the transfer of a certain information. For instance:

- The interaction between components of a communication system (cf. section 3.1);
- The interaction between two colliding rigid bodies (cf. section 4.1).

By that abstraction, the system model receives its hybrid nature: (discrete) *events* are integrated to define certain actions to be performed at a certain time instant. With those events, as opposed to the original physically concurrent interaction, the *causality* of those events has to be introduced. This causality further presents itself as a communication and synchronization problem in the context of computer programming.

The sequential programming technology can handle causality problem easily by applying a global list of unprocessed events, whereas, in concurrent programming, dozens of methods, e.g. semaphores, message passing, monitors, etc., are utilized to solve it.

Since the objects in OO modeling are functionally the same as the parallel processes in the concurrent programming, OO modeling and simulation of physical systems faces similar challenges as concurrent programming, regarding the causality problem. The simulation tool used to evaluate the OO models further defines special disciplines for solving causality problems.

Models arisen from OO modeling have to be firstly interpreted by a simulation tool. The working schema of the interpreter/compiler and the run-time system of the simulation tool play dominant roles in rebuilding the causality. The simulation tool has to decide how to update and process pending event-set dynamically, based

on the information gathered from models. Consequently, the modeler should have solid knowledge about the simulation tool to ensure the correctness of the model.

Figure 2 depicts the causality problem of OO modeling. Supposing the system should execute the operations a, b, c, d in a defined sequence at the time instant t_E: a()→c()→b()→d(). In the normal OO programming case, a top-level program can be defined to manage the calls of methods to achieve this goal. However, in the OO modeling and simulation tool, the lack of upper-level model determines that additional synchronization information is required to define the sequence. Without such information, the simulator itself can merely ensure that all the methods are executed in one possible sequence at the time instant t_E. Formulating the synchronization information must strictly follow the disciplines defined by the simulation tools. In the following section, we will exemplarily exam the simulation tool Modelica® / Dymola® in this aspect.

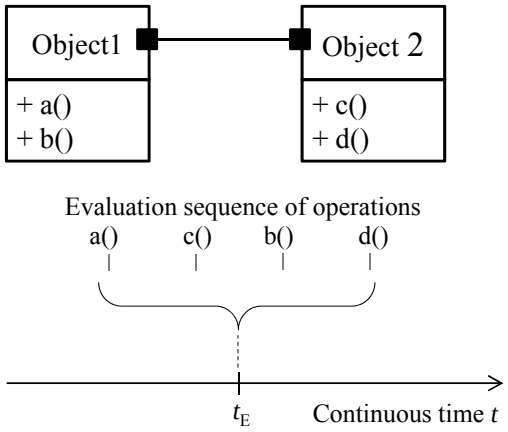

Figure 2. The causality problem in object-oriented modeling

2.2 Causality in an object-oriented modeling and simulation tool

To illustrate the basic working schema of a simulator, the widely used modeling and simulation environment Modelica® / Dymola® is taken as an example. Modelica® is an OO equation-based language, primarily intended for physical system modeling. Additionally, with the support of assignment-based algorithms and special language elements, Modelica® is also capable for modeling hybrid Systems (Elmqvist et al., 2001). Due to the usage of synchronous differential, algebraic and discrete equations as well as automated formula manipulation (Cellier and Elmqvist, 1993) provided by the simulator Dymola®, concurrency and causality assignment problems are solved for equation-based modeling (Mosterman et al., 1998), (Lundvall and Fritzson, 2003), (Otter et al., 1999).

In equation-based models, the equations are solved for unknowns concurrently at any instant of time. In (Otter et al., 1999), the authors demonstrate that, at the instance of discontinuity, the order of equations is determined by dataflow analysis, resulting in an automatic synchronization of continuous and discrete equations. Thereby, deterministic behavior is guaranteed. At the meanwhile, thanks to the single-assignment rule of Modelica®, the so-called algebraic loop problem can be detected by the compiler during translating the models.

The algorithmic sequence, in which unknowns will be solved for, is of no interest in equation-based modeling. However, for modeling computational and communication systems (which are essentially discrete-event systems), it is more straightforward to formulate the behavior in algorithms. Furthermore, algorithms are more preferable as multiple assignments to the same variable are frequently required, which is not allowed by equations due to the single-assignment rule. Modelica® supports algorithm declaration. For ordering multiple events at the same time instant, data-dependency order and declaration-order can be used, where the latter one is not supported in Modelica® language specification (Fritzson, 2004). However, the comparison of the specification and the implementation details in Dymola® reveals some divergences: e.g., the declaration order in a single algorithm section within one model is supported in Dymola®.

The general procedure of defining execution sequence of multiple functions (operations) at the same time instant in Modelica® is as follows: Firstly, each segment of algorithm has to be encapsulated into an event handler. Secondly, the condition of this event handler has to be defined in the body of previous event handler and activated properly. Thus, the series of method calls is then represented by a series of events. However, the execution of the series of events is instantaneous from the viewpoint of continuous simulation. Special support by the simulation tool, named event iteration, is required. *Event iteration* denotes the situation that when an event occurs, new values of a system variable may immediately trigger a further event (Mosterman, 1999). In case of Modelica® / Dymola®, the desired execution order can be modeled by an intentional definition of event priority, event propagation and data dependency order. Though it is possible to use above mentioned techniques to rebuild the sequence of algorithms, it complicates the formulation and verification of models. Especially for modeling large-scale heterogeneous systems, the event propagation and data-dependency ordering are barely manageable due to the large number of interconnected objects. Furthermore, since the algorithms may be coupled, certain issues such as algebraic/algorithmic loop and instantaneous multiple updates emerge.

Moreover, the language specification of Modelica® is to some extent ambiguous concerning the event-servicing algorithm. The author has mentioned in (Fritzson, 2004): "The Modelica language specification does not prescribe any special algorithm [for event-servicing algorithm], thereby giving freedom to the Modelica language im-

plementer to invent even better algorithms." This kind of divergence can be found in the different interpretations of the pre() operator in different tools including Dymola, OpenModelica and SimulationX.

Further in (Nikoukhan, 2007), issues concerning synchronous and asynchronous events in Modelica® and some of its compilers, e.g. Dymola® and Scicos, are discussed. The author shows that different interpretations of the specification may possibly lead to considerable differences in the ways of model construction and compiler implementation.

Last but not least, the OO model structure has to be interpreted and transformed into ordered codes by a front-end compiler. The existing diversity of front-end compilers is another source of uncertainty.

These kinds of vagueness lead to different implementations of simulation tools, and bring about limitations for the user. Firstly, the way of model construction must strictly obey the disciplines drawn by the simulator. Secondly, the correctness of the simulation is strongly dependent on the "quality of implementation".

It is suboptimal if the constructed model has to be modified for using in other tools which differ from the original implementation tool. To avoid this inconvenience, the modeler has to adhere to clearly-defined language specifications rather than "exploiting" ambiguities in a trial-and-error modeling policy.

In the following sections 3 and 4, we will demonstrate the two most important manifestations of the causality problem—algorithmic loops and instantaneous multiple updates—and examine some modeling approaches to handle the causality problem in a systematic manner. Furthermore, the systematic solution benefits from its scalability, i.e., numerous overlapped algorithmic loops and instantaneous multiple updates are solved in the same manner.

3 ALGORITHMIC LOOP

3.1 Exemplary problem description

Similar to the algebraic loop problem that usually results from the circular interconnection of different objects, coupled algorithms may cause loops. In order to distinguish this type of loops from the common algebraic loop, we name it algorithmic loop. An *algorithmic loop* is caused by iterative, mutually dependent events in different objects that are enclosed in a loop. A typical concrete example is the activation and deactivation of the same conditional event. That is, the code inside a conditional event changes the condition that activates the event. While an algebraic loop may be solved by some kind of iteration or elimination algorithm (Cellier and Elmqvist, 1993), there is no automatic method for solving an algorithmic loop. Typically, the simulation tool will give some kind of warning or refuse the compilation of models containing algorithmic loops.

A typical real-world example concerning the modeling of CPS is the Carrier Sense Multiple Access (CSMA) communication illustrated in Figure 3. The transceiver is going to send a message on the medium. Firstly, the transceiver inquires the state of the medium. If the medium is busy, the transceiver keeps waiting; and if the medium is free, it begins transmission. At the beginning of transmission, the medium sets its state as busy in order to block other communication inquiries. After transmission time has expired, the transceiver stops transmission and the medium sets its state to free. Following the OO modeling paradigm, we get two models with coupled algorithms. Since medium_free is the guard condition for transmitting, while transmitting simultaneously forces the reset of the guard condition medium_free, an algorithmic loop is produced.

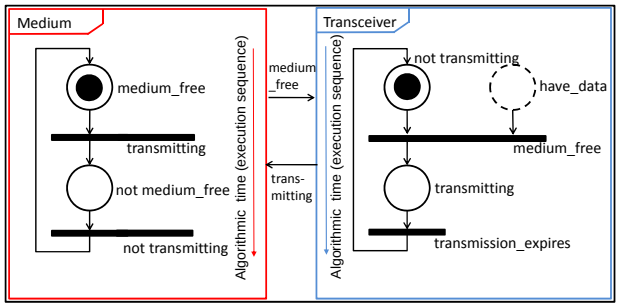

Figure 3. OO model of the CSMA communication problem containing an algorithmic loop

Characteristic **pseudo code** (not Modelica® notation) of the algorithmic loop is shown as follows.

```
model Transceiver
algorithm_to_generate_have_data_event();
when have_data and medium_free then
    transmitting := true;
end when;
end Transceiver;

model Medium
when transmitting then
    medium_free := false;
end when;
end Medium;
```

In the timing diagram in Figure 4, the algorithmic loop problem displays two opposed, simultaneous, coupled step functions. This type of signal changes is very common in the modeling of informational systems if the dynamics of the signal is abstracted by using step function. The resultant model has been tested in the two common simulation software environments Modelica® / Dymola® and Stateflow® / Simulink®. Both demonstrate that the system cannot be simulated without modifying the model or making use of additional language elements.

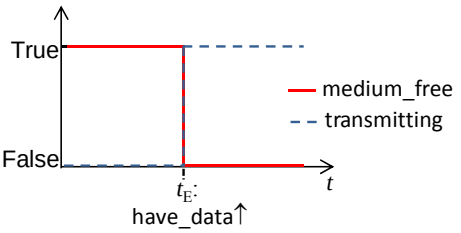

Figure 4. Timing diagram of the algorithmic loop in the CSMA communication model

3.2 Possible solutions

Simplified physical dynamics (continuous approximation): Defining one of the variables of the loop as a continuous state variable builds an intrinsically meaningful causality. Mathematically, this results in a first-order system; the loop is resolved by the state's initial value. The variables medium_free and transmitting are declared as type Real. The respective condition is formulated by means of a threshold. Thereby, the models are rewritten in:

```
model Transceiver
algorithm_to_generate_have_data_event();
when have_data then
    if medium_free > threshold then
        transmitting := true;
    end if;
end when;
end Transceiver;

model Medium
a*der(medium_free)+medium_free=1–transmitting;
end Medium;
```

This approach preserves the OO structure. The main drawback lies in slowing down the simulation speed because of the potential stiffness caused by the continuous function. The parameters a and threshold have to be adjusted to face the dynamic requirements of the discrete variable transmitting. The resultant signals are illustrated in Figure 5.

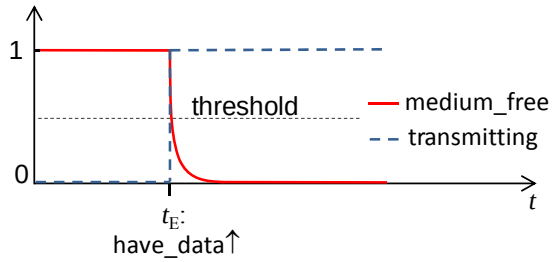

Figure 5. Timing diagram using simplified physical dynamics

Special language support: Modelica® provides the special operator pre() to denote the left limit value $x(t^-)$ of a variable x at time instant t. It is suggested to use this operator to cut algebraic loops. This solution is proposed in most Modelica publications concerning concurrency and algebraic/algorithmic loop problem, e.g. in the work of Lundvall in (Lundvall, 2003) and in the recent Modelica DEVS library ("DEVSLib") (Sanz, 2010). Using the pre() operator, the medium model is modified as follows:

```
model Medium
when pre(transmitting) then
      medium_free:=false;
end when;
end Medium;
```

Recall the timing diagram from Figure 4: At the time instant t_E, the trigger event have_data becomes true. The variable transmitting has the value false at t_E^- and true at t_E^+, which denotes a discontinuous step. The condition pre(transmitting) is not triggered jet. After the trigger event at t_E has been evaluated, the assignment pre(transmitting) := transmitting is executed by the simulator to define the new initial value for the numerical solver. The event handler when pre(transmitting) is then triggered at the exact same time instant t_E. Thanks to the assignment instruction, the two simultaneous events are subsequently handled in two separated event handling processes. The algorithmic loop problem is thereby solved.

The pre() operator is one of the key features in Modelica® for handling algebraic and algorithmic loop problems. By means of when pre(condition), multiple events at the same time instant can be ordered in a sequence. In contrast to a when (condition) clause, which handles the multiple events in a single event iteration procedure, when pre(condition) isolates each event handling in a single event servicing process by means of calculation consistent restart values between them.

However, using the pre() operator is to some extent arbitrary. We can demonstrate this in our example, either using when pre(transmitting), when pre(have_data and medium_free)[1] or even both of them, which will all produce the correct result. This arbitrariness reduces the readability of the code and therefore raises the difficulty of code maintenance and interchange.

Model re-construction: Obviously, flat model code with explicitly defined sequence of algorithms will always resolve the algorithmic loop. However, the surrender of the OO structure is principally contradictory to modern modeling practice.
Considering object-oriented structure as a constraint for model re-construction, only the critical code segments that causes the algorithmic loop has to be re-constructed in the form of a local flat code. In other words, the code segments which are originally distributed in two models are now re-arranged in one single model, where the cau-

[1] It is necessary in Dymola® to form Boolean expression have_data and medium_free to a single Boolean variable in order to use pre() operator .

sality is defined explicitly. This method is shown in the following:

```
model Transceiver
algorithm_to_generate_have_data_event();
observer_interface to monitor transmitting;
end Transceiver;

model Medium
when have_data and medium_free then
    transmitting := true;
end when;
when transmitting then
    medium_free := false;
end when;
end Medium;
```

Thanks to the observer_interface, one model can monitor its variables though the algorithms for manipulating these variables are implemented in another model. This approach merely requires that the simulator supports declaration order in the execution of algorithms. This requirement is easily satisfied because it is the basis of sequential programming technique. Since the algorithm is ordered and integrated in a single segment. It does not need any manipulation by the front-end compiler.

4 INSTANTANEOUS MULTIPLE UPDATES

4.1 Exemplary problem description

In the modeling of computation and communication components, multiple manipulations of the same variable at a time instant are often required. Moreover, temporary values during manipulations may be required for triggering new events. We define the term *instantaneous multiple updates* to denote this problem (Figure 6). This kind of problem is considered as a continuation of the causality problem described in section 3. It raises the following question to the simulator: How to ensure the observability of a temporary value update across model objects?

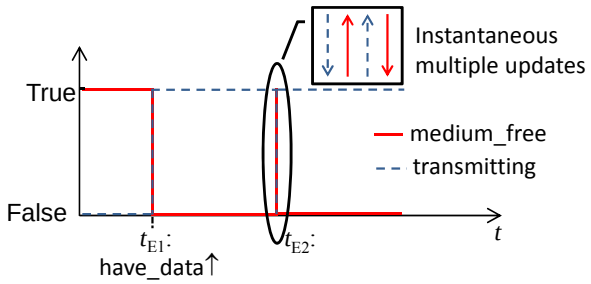

Figure 6. Timing diagram of instantaneous multiple updates in the CSMA communication model

Assuming the Transceiver has cached data to be sent successively. In order to simplify the model and reduce the number of events, the short pause time (network idle time) between two successive transmissions is ignored. At the time instant t_{E2}, four events in a series occur: transmitting↓ ⇒ medium_free↑ ⇒ transmitting↑ ⇒ medium_free↓. Each variable has the same value before and after the event time instant, but the temporary value update between events must be evaluated and handled correctly by the simulator.

The instantaneous multiple updates problem exists not only in modeling informational systems but also in modeling physical systems. Newton's cradle, with its elastic collisions of rigid bodies, is a classic example which shows the effect of handling iterative events in simulation.

Consider the Newton's cradle in Figure 7. It consists of three balls with the same mass *m*. (There will not arise further problems with more than 3 balls.) The initial conditions are $v_1 = v$ and $v_2 = v_3 = 0$. The balls b_2 and b_3 are set directly near each other.

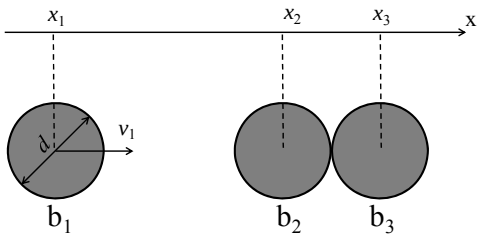

Figure 7. Newton's cradle

By collision, where $x_2 - x_1 \le d$ and $v_1 > v_2$, the momentum and kinetic energy are transferred from b_1 to b_2, and subsequently, from b_2 to b_3. In reality, the exchange between two contacting balls takes a propagation delay which is subject to the speed of sound. However, in modeling practice, the exchange is normally considered as instantaneous. The instantaneous transfer of momentum and energy according to the collision sequence causes iterative events. The iterative events abstract the transient behavior from the real world, which has to be handled correctly in a simulation tool. The velocity of b_2, at the time instant "between" the collisions with b1 and b3, takes a temporary value.

We begin with the conception of objects. A ball object's continuous behavior is described by der(x) = v and der(v) = 0 (without external forces). In order to decide on a collision and to re-initialize its velocity in case of a detected collision, a ball has to get the information from all adjacent balls: their positions, velocities and masses. In this example, since masses and dimensions of balls are the same, it is sufficient to exchange the velocity and position information. For simplifying, only the collision sequence beginning from the leftmost ball (b_1) to the rightmost ball (b_3) is considered. For the opposite direction, respective event definitions have to be added. Pseudo code showing the model object Ball is given below:

```
model Ball
parameter Real initPosition;
parameter Real initVelocity;
Real velocity(start=initVelocity);
Real position(start=initPosition);
equation
der(position)=velocity;
der(velocity)=0;
collision_on_left_side = distance_on_left_side <= 0
and velocity_of_left_ball − velocity > 0;
collision_on_right_side = distance_on_right_side<=0
and velocity − velocity_of_right_ball > 0;
when collision_on_left_side then
    reinit(velocity, previous_velocity_of_left_ball);
end when;
when collision_on_right_side then
    reinit(velocity, previous_velocity_of_right_ball);
end when;
end Ball;
```

Considering the time instant when ball b_1 reaches b_2, there is a series of two collisions: Collision between b_1 and b_2 → Collision between b_2 and b_3. Each collision contains two *synchronous events*: collision_on_right_side in the left ball and collision_on_left_side in the right ball. These events are synchronous because the triggering conditions for them can be traced back to an identical source.

At the time instant of collision, the following pseudo code segments describe the re-initialization from a top-level view. Starting with the collision between b_1 and b_2:

```
when collision_on_right_side_of_b1 then
    reinit(velocity_of_b1, previous_velocity_of_b2);
end when;
when collision_on_left_side_of_b2 then
    reinit(velocity_of_b2, previous_velocity_of_b1);
end when;
```

For the collision with b_2 and b_3:

```
when collision_on_right_side_of_b2 then
    reinit(velocity_of_b2, previous_velocity_of_b3);
end when;
when collision_on_left_side_of_b3 then
    reinit(velocity_of_b3, previous_velocity_of_b2);
end when;
```

As discussed before, since the update of the velocity of b_2 immediately activates the second collision, the two collisions must be handled simultaneously. Consequently, there are four synchronous events at the time t_E of collision. To handle the multiple events at the same time instant, two issues have to be considered:

The first issue is the ordering of the events. As discussed in section 2, there are various approaches capable for doing this. The desired execution sequence of these events to be achieved after ordering is this:

```
when collision then
    reinit(velocity_of_b1, previous_velocity_of_b2);
    reinit(velocity_of_b2, previous_velocity_of_b1);
    reinit(velocity_of_b3, previous_velocity_of_b2);
    reinit(velocity_of_b2, previous_velocity_of_b3);
end when;
```

The second issue concerns the instantaneous multiple updates of the variables in the event handlers. In the example, the term "previous_velocity_of_…" does not necessarily denote the Modelica® operator pre(). It is essentially a verbal expression which describes the previous velocity of a ball before a collision. In the sense of a sequential algorithm, it is the value of a variable which should be retained before manipulating the variable. In the example, the variable previous_velocity_of_b2 should take a transient value according to the above algorithmic execution sequence to ensure the correctness of the result. The transient value is presented as a *Dirac pulse* in the timing diagram of variables with collision at time t_E (Figure 8).

Retaining a variable value is easily done in an algorithm by assigning the value to an auxiliary variable before manipulating it. In the OO equation-based modeling, objects use *connectors* to interchange variables. However, the simulator itself takes charge of exchanging variables via connectors in the background. Since this procedure is hidden for the modeler, it is not possible to order the series of assignments systematically. As illustrated in the code segment above, the variable previous_velocity_of_b2 should be updated after the second reinit(…). However, there is no explicit assignment to update its value on the code level.

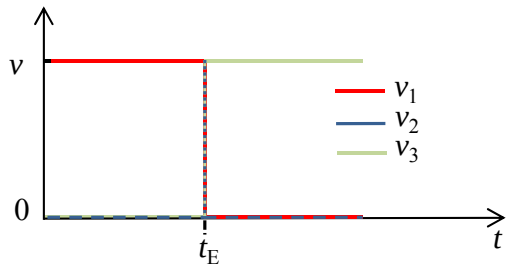

Figure 8. Timing diagram of the intended behavior of Newton's cradle

4.2 Possible solutions

Simplified physical dynamics (continuous approximation): Rather than modeling the instantaneous momentum and energy exchanges between balls, we can use the continuous approximation to describe the collisions (Figure 9).

The ball is then modeled as a spring-mass-system. The stiff spring between balls approximates the elastic deformation upon collisions. The main drawbacks of this

approach are the selection of spring constant and the slow-down of the simulation speed.

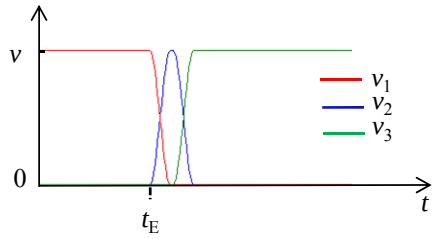

Figure 9. Timing diagram of the balls' velocities in Newton's cradle modeled with continuous approximation

Special language support: According to the analysis in section 3.2, Modelica® / Dymola® will handle the four events caused by one collision in one single event iteration. During the event iteration, the pre(...) value of a variable cannot change and retains the value at the left limit of the event time instant. Thus, the transient value of velocity_of_b2 cannot be presented by the pre() operator in current model construction.

Similar to the solution given in section 3.2, we can use "when pre(condition)" to enforce event iteration and update the value of pre(variable). The modified model code is listed in following:

```
model Ball
⋮
when pre(collision_on_left_side) then
reinit(velocity, pre(velocity_of_left_ball));
end when;
when pre(collision_on_right_side) then
reinit(velocity, pre(velocity_of_right_ball));
end when;
end Ball;
```

This model enforces the re-calculation of variables between the collision series (collision between b1 and b2 → collision between b2 and b3). In other words, the two collisions (events) are not handled in a single event iteration process but separately in two individual steps. This allows the velocity of b2 being transiently updated after the first event handling and thus produces the correct simulation result.

Model re-construction: The instantaneous multiple updates only has a local scope within a model object. In order to ensure the oberservability of the temporary updates, it is preferable to implement the series of events, as long as they are simultaneous, into one single model object. The re-constructed models are listed in the following:

```
model Ball
interface to Collisionmanager;
Equation
der(position)=velocity;
der(velocity)=0;
end Ball;
```

```
model Collisionmanager
n interfaces to n balls;
definition of the collision events as a Boolean array
collision[n-1];
when collision then
   for i in 1:(n–1) loop
     if collision[i] then
     preVelocity[i+1]:=velocity[i+1];
     reinit(velocity[i+1], velocity[i]);
     reinit(velocity[i], preVelocity[i+1]);
     end if;
   end for;
end when;
end Collisionmanager;
```

In this new construction, the Ball model describes its own continuous dynamics between collisions, while the interactive discrete-event behavior (the collisions and resultant re-initializations) are handled in the Collisionmanager model. Based on the explicitly defined declaration order of the statements, the re-initializations of the velocities are performed via auxiliary variables (preVelocity[]) instead of via the pre() operator. This arrangement guarantees a clear view of the expected re-initialization procedures.

5 INTELLIGENCE RE-ALLOCATION

Among the methods discussed in sections 3.2 and 4.2, the *model re-construction* re-structures the code segments (behavioral intelligence of objects) for allowing the modeler to define the causality explicitly. In comparison with the continuous approximation, it highlights the causality declaration and improves the readability of the model codes. Meanwhile, it utilizes no special language elements, which potentially extends the applicability of the model among different simulators. We name this method *intelligence re-allocation* according to its working scheme, which is generalized in Figure 10.

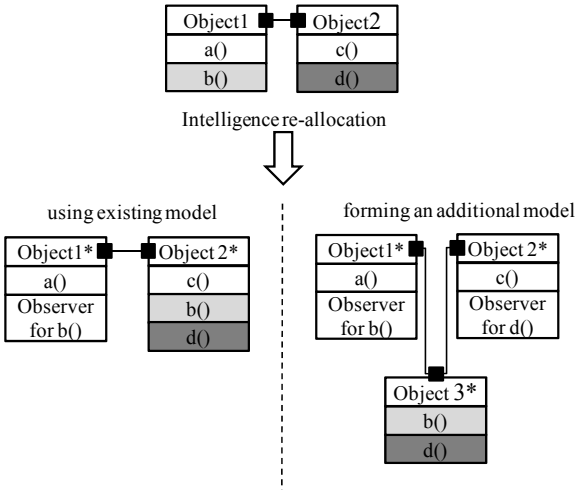

Figure 10. Intelligence re-allocation

Firstly, the code segments are analyzed and categorized in two parts: eigenbehavior intelligence (no shadow) and

causality-related intelligence (shadowed). Subsequently, the causality-related code segments are either re-allocated in one readily available model (lower left part) or in a new model (lower right part). Additionally, these code segments have to be ordered explicitly to build the desired causality b()→d(). Furthermore, for a convenient usage of the models, observer interfaces to monitor the relevant variables are defined in the model objects defining the continuous behavior.

Intelligence re-allocation gives a good compromise between the OO structure and the interpretable causality. On one hand, OO structure is preserved according to carefully code analysis. On the other hand, the causality-related events are clearly ordered by their declaration order within a single model. Since the centralized handling of causality-related code rounds the causality-resolving mechanism provided by certain simulation tools, this modeling scheme can be more easily mastered without solid knowledge about the working principles of the used simulator.

One remaining argument about this approach is that the resultant models are no more strictly conform to the object definition in the computer science, which defines clear scopes of the respective variables and methods. However, in the OO modeling, the structure-conserving principle is preserved by applying appropriate connectors. The connectors (observer interfaces mentioned in Figure 10) extends the scopes of concerned variables, thus the use of the models remains intuitive for end users.

6 SUMMARY AND CONCLUSION

In the simulation of hybrid system, the handling of iterative events must be highly valued. In the object-oriented (OO) modeling paradigm, it is easily error-prone and thus a systematic solution is required. The following two core issues were studied in this paper:

- *Algorithmic loop*, i.e. iterative, mutually dependent events in different objects that are enclosed in a loop, causes problems for which special language support ('pre' operator) is only limitedly suited with respect to model code understandability.
- *Instantaneous multiple updates*, i.e. the value of a variable undergoes multiple updates at an event time instant, while the temporal values may trigger new events across model objects. Handling this problem requires the exact adherence to a specific execution sequence of algorithms which are distributed over diverse objects. However, since the intercommunication of the variables over objects is executed by the simulator in the background, the correct execution sequence cannot be clearly enforced.

We have discussed the applicability of three methods: *Continuous approximation*, the tool-specific *'pre' operator* and *intelligence re-allocation*. Besides these solutions, *model flattening*, which completely sacrifices the advantage of the OO modeling paradigm, is considered to be a valid but antiquated solution. Depending on the structure and size of the respected hybrid system, different solutions can be well-suited, limitedly suited, or unsuited (cf. Table 1). In compact hybrid systems (cHS), the flat-model solution (a.) or the modeling and simulation effort of the parameter-intensive and numerically inefficient solution (b.) can still be feasible. In large-scale models, like CPS, a feasible solution must preserve the OO paradigm to the largest extent possible (as in solution c. and d.).

	Algorithmic loop		Instantaneous multiple updates	
	cHS	CPS	cHS	CPS
a. Flattening of the model	+	o/−	+	o/−
b. Continuous Approximation	o	o/−	o	o/−
c. 'pre' operator (tool-dependent)	+	+	o	o
d. Intelligence re-allocation	+	+	+	+

Solution is: + (well-suited); o (limitedly suited); − (unsuited)

Table 1. Applicability of solutions for causality reconstruction in compact Hybrid Systems (cHS) / Cyber-Physical Systems (CPS).

A precise and conscious use of the 'pre' operator requires a deep understanding of its tool-specific implementation details—far beyond the knowledge necessary to define the "normal", continuous model behavior, which is comparably easy to learn, well documented, and for which OO modeling is widely appreciated.

Among the four solutions given in the survey, *intelligence re-allocation* is the only method which is advantageous for all considered systems and both types of event iteration (*algorithmic loop* and *instantaneous multiple updates*). Due to the clarity and genericity of its structure, this approach achieves the widest applicability and highest scalability. In the implementation of 'NCLib', the solutions c and d have been deployed. Although both of them achieve correct results in the simulation of various heterogeneous CPS, the intelligence re-allocation solution, due to the highlighted causality assignments, has shown remarkable advantage regarding the building, debugging and maintenance of the models.

REFERENCES

Bergero, F., E. Kofman, 2011. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *SIMULATION: Transactions of the Society for Modeling and Simulation International*, vol. 87.

Cellier, F.E., 1979. *Combined Continuous / Disrete System Simulation by Use of Digitl Computers:*

Techniques and Tools. PhD Thesis, ETH, Zurich, Switzerland.

Cellier, F.E., H. Elmqvist, 1993. Automated Formula manipulation Supports Object-Oriented Continuous-System Modeling, *IEEE Control Systems*, 13(2), pp. 28-38.

Derler, P., E.A. Lee, and A.L. Sangiovanni-Vincentelli, 2011. *Addressing Modeling Challenges in Cyber-Physical Systems*. Technical Report, University of California at Berkley, p.12.

Elmqvist, H., S.E. Mattsson, and M. Otter, 2001. Object-Oriented and Hybrid Modeling in Modelica. *Journal Européen des systèmes automatisés*, vol. 35, p. 395-404.

Fritzson, P., 2004. *Principles of Object-oriented modelling and simulation with Modelica 2.1*, Wiley-IEEE Press, ISBN 0-471-471631.

Henriksson, Dan; H. Elmqvist, 2011. Cyber-Physical Systems Modeling and Simulation with Modelica. *In Proc. of the 8th International Modelica Conference*, Dresden, Germany.

Henzinger, T.A., 1996. The theory of hybrid automata. *In Proc. of the 11th Annual Symposium on Logic in Computer Science(LICS)*, IEEE Computer Society Press, 1996, p. 278-292.

Henzinger, T.A., P.-H. Ho, and H. Wong-Toi, 1997. HyTech: A Model Checker for Hybrid Systems. *International Journal on Software Tools for Technology Transfer*, vol. 1, Springer-Verlag, p. 110-122.

Lee, E.A., 2008. *Cyber Physical Systems: Design Challenges*. Technical Report No. UCB/EEECS-2008-8.

Liu, L.; G. Frey, 2008. Feasibility Analysis for Networked Control Systems by Simulation in Modelica. *In Proc. of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008)*, Hamburg, Germany, p. 729-732.

Lundvall, H., P. Fritzson, 2003. Modeling Concurrent Activities and Resource Sharing in Modelica, *In Proc. of The 44th Scandinavian Conference on Simulation and Modeling (SIMS2003)*, Västerås, Sweden.

Mosterman, P.J., M. Otter, H. Elmqvist, 1998. Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica™, *In Proc. of Summer Computer Simulation Conference*, Reno, Nevada, USA, p. 314-319.

Mosterman, P.J., 1999. An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages, *In Proc. of the second International Workshop on Hybrid Systems: Computation and Control*, p. 165-177.

Nikoukhan, R., 2007. Hybrid Dynamics in Modelica: Should all Events be Considered Synchronous. *In Proc. EOOLT Workshop at ECOOP'07*, Berlin, Germany.

Otter, M., H. Elmqvist, S.E. Mattsson, 1999. Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle, *In Proc. of IEEE Symposium on Computer-Aided Control System Design*, Hawaii, USA., p. 151-157.

Reuterswärd, P, J.Åkesson, A.Cervin, K-E.Årzén, 2009. TrueTime Network—A Network Simulation Library for Modelica. *In Proc. of the 7th International Modelica Conference*, Como, Italy, p. 657-662.

Sanz, V., A. Urquia, S. Dormido, 2010. Integrating Parallel DEVS and Equation-Based Object-Oriented Modeling. *In Proc. of the Spring Simulation Multiconference*, Orlando, FL, USA.

Wagner, F.; L. Liu.; G. Frey, 2008. Simulation of Distributed Automation Systems in Modelica. *In Proc. of the 6th International Modelica Conference*, Bielefeld, Germany, Vol. 1, p. 113-122.

Zeigler, B.P., H. Praehofer, and T.G. Kim, 2000. *Theory of Modeling and Simulation*, 2nd ed., New York: Academic Press.