SIMULATION

An extension of the OpenModelica compiler for using Modelica models in a discrete event simulation

James Nutaro

The online version of this article can be found at:

Published by:

⑤SAGE

http://www.sagepublications.com

On behalf of:

Society for Modeling and Simulation International (SCS)

THE SOCIETY FOR
MODELING & SIMULATION
INTERNATIONAL

Additional services and information for *SIMULATION* can be found at:

Email Alerts: http://sim.sagepub.com/cgi/alerts

Subscriptions: http://sim.sagepub.com/subscriptions

Reprints: http://www.sagepub.com/journalsReprints.nav

Permissions: http://www.sagepub.com/journalsPermissions.nav

Citations: http://sim.sagepub.com/content/early/2014/11/03/0037549714554480.refs.html

>> OnlineFirst Version of Record - Nov 3, 2014

What is This?

An extension of the OpenModelica compiler for using Modelica models in a discrete event simulation

James Nutaro

Abstract

This article describes a new back-end and run-time system for the OpenModelica compiler. This new back-end transforms a Modelica model into a module for the adevs discrete event simulation package, thereby extending adevs to encompass complex, hybrid dynamical systems. The new run-time system that has been built within the adevs simulation package supports models with state-events and time-events and that comprise differential-algebraic systems with high index. Although the procedure for effecting this transformation is based on adevs and the Discrete Event System Specification, it can be adapted to any discrete event simulation package.

1. Introduction

It is common to view combining discrete event and continuous models as a problem of integrating distinct types of simulation procedures. This viewpoint is evident when using ''combined'' and ''hybrid'' to describe such simulations as they intermix objects that are otherwise separate. However, these combinations are not unique: each creates a different definition of what is a hybrid model and how it behaves. At best, the relationships between these definitions are difficult to characterize; at worst, they are incompatible. A consequence is that tools for hybrid simulation are difficult to integrate with each other and with other tools for discrete event simulation (see, e.g, the discussion by Carloni et al.[1] and Lee and Zheng[2]).

An alternative viewpoint treats simulation of a model's continuous aspects as a particular discrete event system. This viewpoint is intrinsic to the Discrete Event System Specification (DEVS), which encompasses systems having trajectories that change value a finite number of times in any finite interval.[3] This property is satisfied at the interface between discrete event and continuous systems, and also by numerical algorithms used to simulate continuous models. Consequently, a particular DEVS is defined by every numerical algorithm used to simulate the continuous aspects of a hybrid model. A primary motivation for treating hybrid models in this way is that we may integrate them directly into existing packages for discrete event simulation.

This capability is essential for simulating many types of systems that combine very complex discrete event and continuous time dynamics. A good example of this is the digital control of physical systems through a packet switched communication network. Discrete event simulation packages are almost universally preferred for modeling communication networks. Moreover, these models are most often implemented with general-purpose programming languages (e.g., C++) that can efficiently handle data structures such as graphs, queues, and bit strings, which play prominent roles in packet switched communication. On the other hand, modeling of the physical components is most easily accomplished with special-purpose languages (called continuous system simulation languages) that automatically reduce large sets of equations into a simulatable form, with Modelica being a prominent example of such a language. By translating Modelica models into a discrete event system, these types of very complicated simulations can be created quickly and simulated with accuracy and efficiency.

A prior article formalizes this viewpoint as the split-system approach to simulating hybrid models,[4] and

---

Oak Ridge National Laboratory, Oak Ridge, TN, USA

Corresponding author:
James Nutaro, Oak Ridge National Laboratory, PO Box 2008, MS6085, Oak Ridge, TN 37831-6085, USA.
Email: nutarojj@ornl.gov

specific applications of this viewpoint can be found in the literature.[5–7] The essential abstraction in the split system approach is a collection of functions that operate on a set S of states, set X of input, and set Y of output. The states $s \in S$ may be composed of continuous and discrete variables. There are four functions F, G, A, an L that represent the four central elements in a hybrid system simulation. The function $F : S \times \mathbb{R} \to S$ evolves the continuous part of a state $s \in S$ through an interval of length h while leaving the discrete part fixed. The function $G : S \to \mathbb{R}$ gives the time remaining until a change in the discrete part of s will occur. The function $A: S \times X_\Phi \to S$, where $X_\Phi = X \cup \{\Phi\}$, describes the change in s that occurs due to a discrete input $x \in X$ or in response to an internal event Φ, with the latter occurring at the instants $G(s) = 0$. The function $L: S \to Y$ gives the output of the system when it is in state s.

While these elements appear in all simulation software for hybrid systems, their arrangement is typically in a loop that may be stylized as follows.

1. Advance the state from $s(t)$ to $s(t + h)$ using F and a step size h selected for numerical accuracy, stability, and to avoid G going too far past zero.
2. Set $t \leftarrow t + h$.
3. If $G(s(t))$ is at or has passed zero, then change the state from $s(t)$ to $A(s(t),\Phi)$.
4. Calculate the output $L(s(t))$.
5. Repeat.

This arrangement is natural when the purpose of the simulator is to calculate trajectories for a specific model in isolation, but it is very inconvenient if we want to use that model as part of a larger, overarching simulation. Nonetheless, this arrangement is often so deeply ingrained into the simulation software that large modifications are infeasible, and this leads to undesirable trade-offs between accuracy, numerical stability, and execution time (see, e.g., the discussion of step size selection in EPOCHS[8] and the review of co-simulation by Broman et al.[9]).

To overcome this problem, the split system approach begins with the assumption that the hybrid model will be used as a component within a larger, overarching simulation system and arranges F, G, A, and L for this purpose. This new arrangement defines an atomic Parallel DEVS model[3] (Parallel DEVS is assumed throughout this article and so the Parallel adjective is dropped for brevity) in the form

$$\delta_{int}(s) = A(F(s, ta(s)), \Phi) \qquad (1)$$

$$\delta_{ext}(s, e, x) = A(F(s, e), x) \qquad (2)$$

$$\delta_{con}(s, x) = A(F(s, ta(s)), x) \qquad (3)$$

$$ta(s) = G(s) \qquad (4)$$

$$\lambda(s) = L(F(s, ta(s))) \qquad (5)$$

An advantage of this approach is that it becomes possible to efficiently and accurately manage time in co-simulations. This atomic model can be used directly as a component within discrete event simulation packages such as OMNET++, OPNET, NS3, adevs, DEVSJAVA, and others, and it can readily participate in federated simulations organized around middle-ware such as the IEEE high-level architecture (HLA).

This article takes an important step towards making the split-system approach suitable for simulations of complex, high-index DAEs by creating a new extension of the OpenModelica compiler[10,11] and a run-time system to support this extension. The extended compiler generates C++ source code for a DEVS atomic model that is intended for use with the adevs simulation package. This atomic model is a C++ class that can be sub-classified to add capabilities for interacting with other components in the discrete event simulation. The source code for this generated C++ class and any sub-classes derived from it can then be compiled and linked into any adevs-based simulation program. For this purpose, the adevs simulation library has been extended with numerical algorithms for continuous system simulation: these include algorithms for locating state events and solving high-index DAEs using the method of dummy derivatives. In the future, the simple numerical procedures used in this incarnation of the run-time system will be replaced with more robust alternatives.

The presentation of the extended compiler and new run-time system is structured as follows. Section 2 contrasts the proposed approach with related work on combining DEVS and Modelica models. Section 3 describes the simulation procedure for combined models and how the extended compiler and new run-time system support this procedure. A detailed description of how a Modelica model is transformed into a DEVS is the subject of Section 4. Section 5 applies this procedure to simulate a robotic arm controlled through an Ethernet network. The article concludes in Section 6 with a brief comparison with other techniques for the co-simulation of discrete event and continuous models.

2. Related work

Prior work on combining Modelica and DEVS models has pursued two distinct approaches. One of these approaches is to extend the Modelica language and its run-time system to facilitate the construction and simulation of discrete event models. Examples of this approach can be found in the literature,[12,13] and these share the goal of creating a single, integrated environment for modeling discrete event and continuous systems. Consistent with this goal, the chief characteristic of this approach is that the combined Modelica/DEVS model is simulated with a loop like that described in Section 1. Consequently, this approach to

integrating Modelica and DEVS models does not facilitate the reuse of Modelica models within existing software for discrete event simulation.

The other approach to combining Modelica and DEVS models is similar to what is proposed here. In this approach, the Modelica compiler and run-time system are modified to produce a module for an existing discrete event simulation package. This type of modification is described by D'Abreu and Wainer,[14] Floros et al.,[15,16] and Bergero et. al.[17] In those cases, the resulting DEVS model uses quantized state integrators to solve the Modelica model's continuous equations with the purpose of reducing the execution time of a simulation, although the same DEVS model could also be used as a component within an overarching, DEVS-based model.

However, because the performance gains obtained by these QSS-based tools depends on an explicit representation of the model's dependency graph, QSS methods have a limited capability to simulate DAE with high index. This limitation manifests itself when the method of dummy derivatives is used to reduce a high-index DAE to a DAE with index 1. This limitation is significant because of the numerical advantages of the dummy derivatives method and its consequent use by OpenModelica and many other simulation packages (see Mattsson and Söderland[18]; a good example of the method can be found in Mattsson et al.[19]).

A brief review of the method of dummy derivatives shows how it poses a difficult, and perhaps intractable, problem for QSS-based simulators. The method has two parts. In the first part, the high-index DAE is reduced via symbolic manipulations to a family of index 1 DAEs. Each DAE in this family has a distinct set of state and algebraic variables, and consequently a distinct dependency graph for those variables. In addition to this family of systems, Jacobian matrices are generated such that their numerical conditioning determines which system to use at each instant of time. This part of the method is done by the Modelica compiler as it parses the model and generates simulation code.

In the second part, the run-time system uses the Jacobian matrix to select a system for use at each step in the simulation. Before each step of the integration procedure, the simulator calculates the Jacobian matrices using the current state of the model and then performs a full pivot of these matrices to determine which of the family of DAEs is best conditioned. This best conditioned system is used in the next integration step.

To use the method of dummy derivatives in a QSS-based simulation would require two significant advances. The first is an efficient, but explicit, coding of the dependency graphs for the family of DAEs. The size of this family grows combinatorially in the number of required and candidate state variables, suggesting that a brute force encoding of each member in a large family will quickly use up the available computer memory. This

problem is avoided by numerical methods that do not require an explicit representation of the model's dependency graph.

The second significant advancement is an alternative to full pivoting of the Jacobian matrices. The full pivot is an $O(N^3)$ operation in the number of candidate state variables, and it is performed each time a state variable is updated. Hence, the otherwise rapid updates of the QSS integrators must now incur the high cost of this pivot operation. Moreover, it has been shown that the time advance function of the resultant of a first-order QSS network is bounded from above by the stable step size of the explicit Euler integrator.[20] So while the QSS integrators perform fewer function evaluations overall relative to a synchronous method, the QSS method generally has more events than the synchronous method has time steps. This suggests that there will be many more pivot operations in a QSS simulation than in a simulation that updates its state variables synchronously. Consequently, the QSS method may lose its advantage of computational efficiency.

The extended compiler described in this article differs from these prior approaches to combining DEVS and Modelica models in two important ways. The first difference is our aim to transform Modelica models into modules for an existing discrete event simulation. Unlike prior work to extend Modelica with a capacity for discrete event simulation, the C++ class generated by our extension to the OpenModelica compiler is not intended for use on its own. Rather, this class is to be sub-classified, the derived class extended for exchanging data with other discrete event models, and then compiled into a discrete event simulation program that uses the adevs simulation library.

The second difference is that the run-time system has full support for simulating high-index DAEs using the method of dummy derivatives. For this purpose, we use the symbolic part of the dummy derivatives method that is already built into the OpenModelica compiler. In the run-time system we use a full pivot of the generated Jacobian to select the best conditioned system of equations at each integration step or event. This straightforward solution is possibly because the new run-time system uses a synchronously updating method to solve the model's continuous equations. In principle, the proposed approach can be realized with numerical solvers such as DASRT that are widely used in Modelica-based simulation tools. This is attractive because it suggests that existing Modelica tools could be quickly adapted to generate discrete event models. Nonetheless, here we demonstrate a Modelica compiler for the split system approach in a much simpler setting that uses the bisection method for locating state events (i.e., to realize G) and an explicit Runge–Kutta method to advance the state of the continuous variables (i.e., to realize F). The functions A and L are intrinsic to the DEVS formulation of the Modelica model.

3. Combining Modelica and DEVS

The Modelica models considered here comprise a vector $\mathbf{q}$ of continuously evolving state variables, a vector $\mathbf{w}$ of continuously evolving algebraic variables, and a vector $\mathbf{z}$ of discretely evolving variables. The discrete variables change value only at discrete instants in time; that is, in response to discrete events. These events may be discrete input to the model or the satisfaction of a logical predicate over the continuous and discrete variables. Between discrete events, the continuous variables evolve according to

$$\dot{\mathbf{q}} = f(\mathbf{q}, \mathbf{w}, \mathbf{z}) \qquad (6)$$

$$g(\mathbf{q}, \mathbf{w}, \mathbf{z}) = 0 \qquad (7)$$

The extended OpenModelica compiler generates a DEVS atomic model that solves these equations using KINSOL for g and a Runge-Kutta integration algorithm for f (see Brenan et al.[21] and Hindmarsh et al.[22]). Previous work[4,5] gives an abstract definition of this DEVS atomic model. To realize this abstract definition, the OpenModelica compiler, in conjunction with its run-time support in the adevs simulation package, produces the five parts of a DEVS atomic model.[3,5]

1. The time advance function *ta*, which gives the smaller of the time remaining until the model's next internal event and the next update of its continuous variables. The next internal event occurs at the earliest instant when any of the model's logical predicates are satisfied. The next update of the continuous variables occurs at the next step of the numerical integrator.
2. The output function λ, which maps the model's present state into a bag of discrete outputs. These outputs are generated immediately prior to every internal event.
3. The internal state transition function δ_{int}, which maps the model's present state into a next state. This function calculates the model's continuous trajectory between discrete events and calculates new values for discrete and continuous variables at internal events.
4. The external state transition function δ_{ext}, which maps the model's present state, the time that has elapsed since the most recent discrete event, and a bag of discrete inputs into a next state. This function calculates new values for the model's discrete and continuous variables upon receiving input from some other discrete event model. The arrival of discrete inputs is called an external event.
5. The confluent state transition function δ_{con}, which maps the model's present state and a bag of discrete inputs into a next state. This function calculates new values for the model's variables when an internal and external event coincide in time.

The simulator for an atomic model has three tasks. These are (i) determine the time of the next event, (ii) calculate the output at that time, and (iii) calculate the new state at that time. Let t_0 be the time of the model's most recent event, t_N the time of its next event, s the state of the model at t_0, t_x the time of the next input (i.e., if input occurs at $t_1 \leqslant t_2 \leqslant \ldots$, then t_x is t_1 until that input is processed, then t_x is t_2 until that input is processed, and so on) and x the input at time t_x. The simulation procedure, which is implemented by the adevs simulation package, has four steps.[3,5]

1. Find the time t_N of the next event. This time is the smaller of $t_0 + ta(s)$ and t_x.
2. Calculate the model's output at t_N. If $t_N = t_0 + ta(s)$, then the output at time t_N is $\lambda(s)$. Otherwise, the model does not generate output at t_N.
3. Calculate the model's state at t_N. The next state is $\delta_{int}(s)$ if $t_0 + ta(s) < t_x$; it is $\delta_{ext}(s, t_N - t_0, x)$ if and $t_x < t_0 + ta(s)$; and it is $\delta_{con}(s,x)$ if $t_0 + ta(s) = t_x$.
4. Set t_0 to t_N and repeat from step 1.

Simulations that combine a Modelica and DEVS model use this simulation procedure for all components of the combined model, both continuous and discrete event. To accomplish this, it is sufficient to generate time advance, output, and state transition functions for the continuous model or, more properly, for its numerical simulator. The modified OpenModelica compiler is used for this purpose.

Figure 1 illustrates how the modified compiler is used to create a simulation program. This simulation program has two parts: source code in the Modelica language, which describes the program's Modelica models, and C++ source code that implements its adevs models. The extended compiler translates the Modelica source code into C++ source code in three steps as follows.

1. Parsing the Modelica models. This is done by the front-end of the compiler, which is not modified for the split system method.
2. Flattening, index reduction, equation sorting, and other symbolic manipulations. This is done by the middle part of the compiler to generate a family of systems in the form of Equations 6–7, Jacobian matrices for selecting which member of the family to use, and expressions for time and state events. This part of the compiler is also unmodified.
3. C++ source code is emitted by the back-end of the compiler, which has been modified specifically to translate the equations provided by the middle part of the compiler into a DEVS model. The emitted source code is an atomic model with its time advance, state transition, and output functions. However, the generated code does not
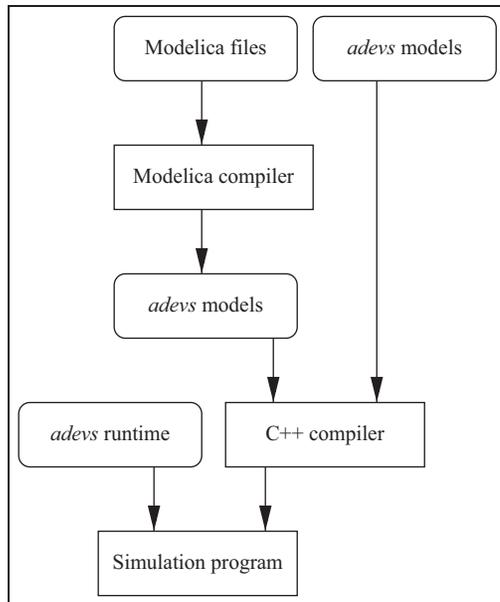
Figure 1. Using the Modelica compiler to create part of a simulation program.

contain instructions for interacting with other DEVS models.

The source code emitted in step 3 is not a complete simulation program because it does not interact with other discrete event models. Instructions for these interactions are added by creating a sub-class that extends the internal, external, and confluent state transition functions, time advance function, and output function generated by the Modelica compiler. This sub-class converts incoming adevs events into data suitable for the Modelica model (e.g., to extract the payload from a network packet and convert it into a floating point number), and it transforms events and data within the Modelica model into events suitable for sending to other adevs models (e.g., to put a sensor measurement into a network packet).

Instances of the DEVS model created by the OpenModelica compiler and sub-classes of this model are used in a simulation by attaching it to an instance of the Hybrid class. The Hybrid class is a part of the adevs simulation library that serves as the run-time system for Modelica models. The Hybrid class is a type of Atomic model that implements numerical procedures for solving the differential algebraic equations using the method of dummy derivatives, for locating state events, and for initializing the model state. The combination of the class generated by the OpenModelica compiler, the class derived from this by the modeler, and the Hybrid class constitute a complete realization of the atomic model defined by the split system method.[4,5]

Figure 2 illustrates the relationships between the Hybrid class, the class generated by the OpenModelica compiler,
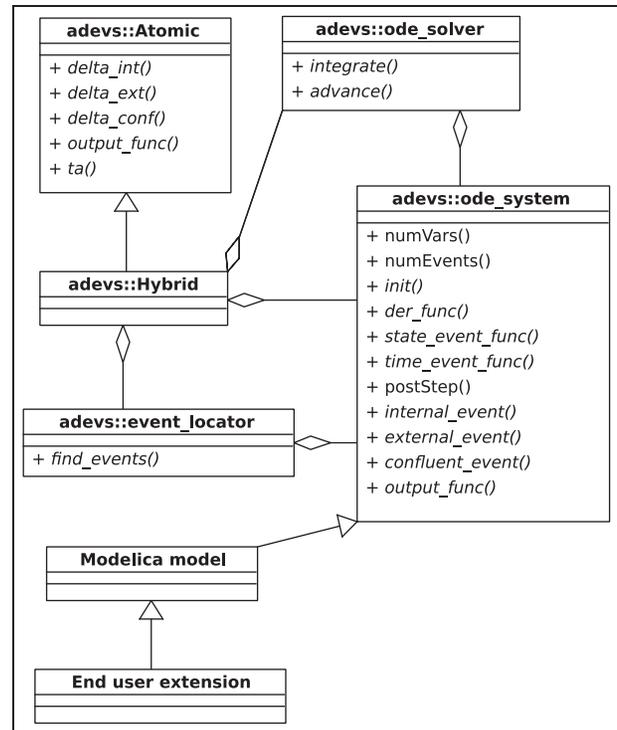
Figure 2. UML diagram of the adevs objects that implement a Modelica model.

and its sub-class that is created by the modeler. The interaction of these classes is described by Nutaro.[5] In what follows, the assembly shown in Figure 2 is treated as a single, atomic model without reference to its components. From this perspective we describe the essential aspects of the assembly: its time advance, output, and state transition functions. In this description, behaviors added by the sub-class are referred to as sub-class code, sub-class functions, and so forth.

4. The Modelica model as an Atomic DEVS

The time advance function is what most distinguishes the extended OpenModelica compiler and new run-time system from prior work on combining DEVS and Modelica models. Therefore, Section 4 begins by defining this time advance function. Definitions of the state transition functions and output function follow naturally from the time advance and a suitable adaptation of previous work on simulating Modelica models.[19,23–25]

4.1. Time advance of a Modelica model

As before, let t_0 be the time of the model's most recent event and s its state, which includes $\mathbf{q}$, $\mathbf{w}$, and $\mathbf{z}$. The time $ta(s)$ until the next internal event at time $t_0 + ta(s)$ is the

smaller of three values. The first value is the preferred step size of the numerical method that advances **q** and **w** between events. This step size is the smallest that satisfies a tolerance for error in the numerical solution and a maximum step size, which are both specified by the modeler. Let h_1 be the step size selected to satisfy these criteria.

The second value used to calculate the time advance is the time remaining until the next internal event for which the time of occurrence is known explicitly. These events are called time events, and the sample function of the Modelica language provides an example. The sample function has two arguments that describe instants when values for the continuous variables must be calculated. These arguments are a start time t and sample interval Δt. Hence, an event occurs at the instants $t + n\Delta t$, $n \geqslant 0$. Let h_2 denote the time remaining to the next time event.

The third value used to calculate the time advance is the time remaining until the next internal event for which the time of occurrence is only known implicitly. These events are called state events and they are due to logical predicates over continuous state variables. The compiler transforms these predicates into zero-crossing functions that change sign when the predicate changes value. The next state event happens at the smallest root t' of the model's zero-crossing functions in the interval $[t_0, t_0 + \min\{h_1, h_2\}]$. If such a root exists, then the time remaining to this event is $h_3 = t' - t_0$. If there is no such root, then $h_3 = \infty$.

The value of h_1 may be calculated using well-known techniques for selecting the step size of a numerical integrator.[26] The value of h_2 is a simple function of the model's current state and the current time. To determine h_3 requires two things: a definition of the zero-crossing functions and a method for finding their roots. Modelica has seven primary operators that generate state events. These are the relations $>$, $\geqslant$, $<$, and $\leqslant$ and the functions floor, ceiling, and div. All other logical predicates (i.e., if, when, and other conditional statements) that generate state events are expressed using combinations of these operators and discrete auxiliary variables.[23,24]

The OpenModelica compiler defers realization of these primary operators to the simulation run-time system, which must provide specific definitions. The definitions used by the adevs run-time system are described below, and these are sufficient to realize if and when statements; the floor, ceiling, div, mod, integer, and rem functions; and comparisons. Zero-crossing functions may also be supplied by subclasses of the DEVS model.

4.1.1. Hysteresis in event detection.

It is intuitively appealing to have the compiler create a single zero-crossing function z for each relation and function in the model that generates state events. This function would be defined such that $z = 0$ at the event and $z \neq 0$ otherwise. However, this single

function presents a serious difficulty to numerical simulation, as illustrated by the model show below.

```
class Zeno
  Real x(start = 0);
equation
  der(x) = if (x < 1) then 1 else 0;
end Zeno;
```

This model is simple enough that its intended behavior is apparent without simulation. The model has a single state variable x that begins at $x(0) = 0$ and, because of the if statement, with $dx(0)/dt = 1$. The model has a state event, encoded by the if statement, that occurs when $x(t) = 1$ and this happens at $t = 1$. At this instant, the derivative switches from $dx/dt = 1$ to $dx/dt = 0$, and the model should remain at $x(t) = 1$ for all $t \geqslant 1$.

However, a simulation that uses the zero crossing function $z(t) = x(t) - 1$ to test for "if $(x < 1)$" will not behave as it should. At $t = 1$, $x(1) = 1$ and, therefore, $z(1) = 0$. Hence, the if statement is triggered and this changes dx/dt to 0 but leaves $x = 1$. When the simulator tests again if there are events to execute at $t = 1$, it discovers that $z(1) = 0$ is still true and so again sets dx/dt to 0. Indeed, the simulation clock will never advance past $t = 1$ because the event indicated by "if $(x < 1)$" is triggered again, and again, and again into perpetuity. This model with this zero-crossing is illegitimate[3,27] (also called a Zeno hybrid system[28,29]). It produces an infinite number of discrete events in a finite interval of time.

A solution to this problem is to apply a hysteresis ϵ to the zero-crossing function. The simplest use of this hysteresis is to find where $z(t) = 0$ and then activate the event some small time ϵ after t. Applying this to the model above, we detect the state event at $z(1) = 0$ and apply the event at $z(1 + \epsilon) = 1 - x(1 + \epsilon) = 1 - (1 + \epsilon) = -\epsilon$. This solves the illegitimacy problem, but introduces a new difficulty. Suppose we replace Zeno with the model shown below.

```
class Zeno2
  Real x(start = 0);
equation
  der(x) = if (x < 1) then 2 else 0;
end Zeno2;
```

Simulating this model, we find the event at $t = 0.5$ where $z(0.5) = 0$. However, the event is applied at $t = 0.5 + \epsilon$ and we have $z(0.5 + \epsilon) = 1 - x(0.5 + \epsilon) = 1 - 2(0.5 - \epsilon) = -2\epsilon$. By doubling the derivative, we have doubled the error in z at the time of the event. Indeed, a bound on the error cannot be determined without knowledge of the derivative of z, which typically is not known prior to simulating the model.

This new problem and the illegitimacy problem are both solved with a more complicated, but more robust, use

of the hysteresis. With this approach, ϵ splits z into two functions $z_1 = z$ and $z_2 = z - \epsilon$. The function z_1 is initially active if $x < 1$ and z_2 otherwise. The active function is switched at each event. In this way, the error in z at the application of the event is controlled explicitly by the algorithm that detects when z_1 and z_2 cross zero. Applying this approach to the model Zeno gives the pair of zero-crossing functions

$$z_1(t) = 1 - x(t) \tag{8}$$

$$z_2(t) = 1 - x(t) - \epsilon \tag{9}$$

Only z_1 is active at the start of the simulation. As before, at $t = 1$, $z_1(1) = 0$ and the event occurs to set $dx/dt = 0$ while leaving $x = 1$. This event also replaces z_1 with z_2. Now at $t = 1$, $z_2(1) = -\epsilon$ and the next event occurs at $t = \infty$. Hence, the model is legitimate and behaves as expected: it reaches its maximum value at 1 and remains there. Applying this model to Zeno2 yields a similar result, with $z_1(0.5) = 0$ and $z_2(0.5) = -\epsilon$. The outcome is independent of the derivative of z.

4.1.2. Comparisons.
Comparisons that generate state events in a Modelica model take the form $e_1 < e_2$, $e_1 \leqslant e_2$, $e_1 > e_2$, and $e_1 \geqslant e_2$ where e_1 and e_2 are expressions that evaluate to a fixed or continuously evolving quantity. For each such comparison, the compiler generates a binary variable z_c that is true if the comparison is true and false if the comparison is false. The compiler also generates functions $z_<$ for $<$ and $\leqslant$ and $z_>$ for $>$ and $\geqslant$ that cross zero when the variable z_c should change value. A hysteresis ϵ is used to avoid the model becoming stuck at an event.

The functions $z_<$ and $z_>$ are defined by

$$z < (e_1, e_2, z_c) = \begin{cases} e_2 - e_1 & \text{if } z_c = \text{true} \\ e_2 - e_1 - \epsilon & \text{if } z_c = \text{false} \end{cases} \tag{10}$$

$$z > (e_1, e_2, z_c) = z < (e_2, e_1, z_c) \tag{11}$$

These definitions cause a true to false transition at zero and a false to true transition at ϵ. Moreover, the following relations are true when $\epsilon = 0$.

$$(z < (e_1, e_2, z_c) < 0 \text{ and } z_c) \text{ or } (z < (e_1, e_2, z_c) > 0 \\ \text{and not } z_c) \Leftrightarrow z_c \neq e_1 \leqslant e_2 \tag{12}$$

$$(z > (e_1, e_2, z_c) < 0 \text{ and } z_c) \text{ or } (z > (e_1, e_2, z_c) > 0 \\ \text{and not } z_c) \Leftrightarrow z_c \neq e_1 \geqslant e_2 \tag{13}$$

These facts are used to calculate the response of the model to discrete events (see Section 4.3). The initial value for z_c is calculated using the initial values of e_1 and e_2. Hence, starting values for $<$ and $\leqslant$ are $z_c(0) = e_1(0) < e_2(0)$ and $z_c(0) = e_1(0) \leqslant e_2(0)$, respectively. Starting values for $>$ and $\geqslant$ are determined in the same manner.

4.1.3. Event-generating functions.
The floor, ceiling, and div functions are similarly defined by using a hysteresis value and auxiliary variable. In this case, the auxiliary variable is an integer z_i that retains the value of the floor, ceiling, or div function at the time of the most recent event. Each instance of a floor, ceiling, or div function generates two zero-crossing functions: one for deciding when z_i increases and another for deciding when z_i decreases. These zero-crossing functions are z_{fu} and z_{fd} for the floor function, z_{cu} and z_{cd} for the ceiling function, and z_{du} and z_{dd} for the div function.

Given an expression e and value z_i, the zero-crossing functions are defined by

$$z_{fu}(e, z_i) = z_i + 1 - e \tag{14}$$

$$z_{fd}(e, z_i) = e - z_i + \epsilon \tag{15}$$

$$z_{cu}(e, z_i) = z_i + \epsilon - e \tag{16}$$

$$z_{cd}(e, z_i) = e - z_i + 1 \tag{17}$$

$$z_{du}(e, z_i) = \begin{cases} z_i + 1 - e & \text{if } z_i \geqslant 1 \\ z_i + \epsilon - e & \text{if } z_i \leqslant -1 \\ 1 - e & \text{if } z_i = 0 \end{cases} \tag{18}$$

$$z_{dd}(e, z_i) = \begin{cases} e - z_i + \epsilon & \text{if } z_i \geqslant 1 \\ e - z_i + 1 & \text{if } z_i \leqslant -1 \\ e + 1 & \text{if } z_i = 0 \end{cases} \tag{19}$$

These functions satisfy the following relations when $\epsilon = 0$.

$$z_{fu}(e, z_i) < 0 \text{ or } z_{fd}(e, z_i) < 0 \Leftrightarrow z_i \neq \text{floor}(e) \tag{20}$$

$$z_{cu}(e, z_i) < 0 \text{ or } z_{cd}(e, z_i) < 0 \Leftrightarrow z_i \neq \text{ceiling}(e) \tag{21}$$

$$z_{du}(e, z_i) < 0 \text{ or } z_{dd}(e, z_i) < 0 \Leftrightarrow z_i \neq \text{div}(e) \tag{22}$$

As with the relations, these facts are used to calculate the response of the model to discrete events (see Section 4.3). The initial value for z_i is calculated from the initial value of e so that for the floor function

$$z_i(0) = \text{floor}(e(0)) \tag{23}$$

and for the ceiling function

$$z_i(0) = \text{ceil}(e(0)) \tag{24}$$

and for the div function

$$z_i(0) = \text{trunc}(e(0)) \tag{25}$$

where trunc rounds its argument toward zero.

4.1.4. Sub-class functions. Zero-crossing functions that are supplied by the sub-class must satisfy two criteria. First, they must be continuous to support algorithms for event detection that assume continuity in the zero-crossing functions. Second, the zero-crossing functions must not make the model illegitimate.

4.1.5. Locating zero-crossings. Any root finding algorithm can be used to locate states events in time by finding the smallest time in the interval $[t_0, t_0 + \min\{h_1, h_2\}]$ where a zero-crossing function $z(l(t)) \approx 0$. The function z may be a sub-class function or any of the zero-crossing functions defined in Sections 4.1.2 and 4.1.3. The argument $l(t)$ may be $e(t)$, z_i; $e_1(t)$, $e_2(t)$, z_c; or $e_2(t)$, $e_1(t)$, z_c as appropriate. During the search for a zero-crossing, the variables in **z**, which include the discrete auxiliary variables, are kept constant at their values for time t_0. Whatever specific root finding algorithm is used, the outcome is two values. The first is a Boolean value indicating the presence or absence of a state event in the interval $[t_0 + \min\{h_1, h_2\}]$ and the second is the h_3 where this event occurs (or ∞ if there is no such event).

The bisection algorithm for N zero-crossing functions in the form $z_k(l_k(t))$, $k \in [1, N]$ is summarized as Algorithm 1. The presence or absence of a state event at h_3 is indicated with the boolean variable atEvent. The

zero-crossings that triggered the event are indicated by boolean flags γ_k, $k \in [1, N]$ such that γ_k is true if z_k generates the state event and is false otherwise.

Algorithm 1 illuminates two problems that are specific to the split system method and do not occur in stand-alone simulations. The first problem is that state events can be lost when the DEVS model responds to input. From this perspective, the key feature of Algorithm 1 is the error tolerance p, which is a necessary component of all root finding algorithms. To see this problem, suppose at time t we have $z(t) = 0$. Owing to p it is possible for the simulator to misplace this state event, believing instead that it occurs later at time $t + \tau$ with $|z(t + \tau)| < p$. Hence, if an input arrives in the interval $[t, t + \tau)$, the simulator may overlook the state event at t. This problem must be corrected for in the model's the external transition function (see Section 4.4).

The second problem is that input may cause a state transition before the time advance expires. Algorithm 1 addressed this possibility by fixing the bisection search on the left-hand side. This retains the model state at time t to calculate its response to input arriving in the interval $[t, t + ta(s)]$. This change to the standard bisection algorithm has a negative impact on performance because it closes in on the root from just the right-hand side. A similar problem will be encountered regardless of the root finding algorithm that we choose. On the other hand, if we wanted to allow for updates of the left- and right-hand sides, then it would be necessary to retain a copy of the state at time t and thereby double the memory required to simulate the model.

4.2. Output function

The output function is invoked just prior to an invocation of the internal or confluent transition function. Recall that the output function is defined by the split system method to be $\lambda(F(s, ta(s)))$ and so requires $s(t)$ at the time of the output. Conceptually, this is accomplished by computing F in the output function but not assigning the result of this computation to the model's state variables. In this way, the state of the model is not modified by the output function.

In practice, there are two choices for calculating the output function. The first choice is to retain the last trial solution for s that was calculated as part of obtaining $ta(s)$. This trial solution is the value of s at $t + ta(s)$, and so it is the $F(s, ta(s))$ needed in the output function. To use the trial solution requires that we keep in memory two copies of the model's variables: one copy for the state at the previous event and one copy that is the trial solution. The advantage of this is that we avoid recalculating F. The second choice is to discard the trial solution and recalculate F when it is needed by the output function. This saves memory, but increases the execution time of the simulation. The simulator presented here uses the first choice.

ALGORITHM 1: Locating state events by bisection.

> **Data**: t_0, h_1, h_2
> **Result**: h_3, atEvent, γ_1, …, γ_N
> $h_3 \leftarrow \min\{h_1, h_2\}$
> **repeat**
>   missedEvent $\leftarrow$ false
>   atEvent $\leftarrow$ false
>   **for** $k \in [1, N]$ **do**
> $\gamma_k \leftarrow$ false
>   **if** *sign* $(z_k(l_k(t_0 + h_3))) \neq$ sign$(z_k(l_k(t_0)))$ **then**
>   **if** $|z_k(l_k(t_0 + h_3))| < p$ **then**
> $\gamma_k \leftarrow$ true
>   atEvent $\leftarrow$ true
>   **else**
>   missedEvent $\leftarrow$ true
>   **end**
>   **end**
>   **end**
>   **if** missedEvent **then**
> $h_3 \leftarrow h_3/2$
>   **else**
>   **return** h_3, atEvent, γ_1, …, γ_N
>   **end**
> **until** *forever*

4.3. Internal and confluent transition functions

The internal transition function is used to compute a new state for the model when the time advance expires without any interrupting input. Let t_0 be the time of the model's most recent change in state and h the value of the time advance so that the current simulation time is $t_0 + h$. The internal transition function calculates the new state of the model at $t_0 + h$ with an algorithm that has four steps.

The first step is to advance the continuous variables **q** and **w** from time t_0 to time $t_0 + h$. This is done by integrating the continuous state variables and solving for continuous algebraic variables using the new values of the continuous state variables.

The second step is to reconsider which DAE system, selected from the family of options created by the symbolic part of the dummy derivatives method, to use at the next simulation event. This is done by calculating and pivoting the Jacobian matrices that were generated by the OpenModelica compiler for this purpose. If a new system is indicated by the pivot operation, then the new choice of state and algebraic variables is recalculated to be consistent with this selection.

The third step is to apply the discrete events that occur at $t_0 + h$. If there are events at this time, then the internal transition function calculates their effect on **q**, **w**, and **z**. This calculation begins by reinitializing all of the discrete variables in **z** that depend on the model's zero-crossing functions. This is done using the comparison operator or discrete function to which each discrete variable is attached, i.e., if z_c is the auxiliary variable for $e_1 < e_2$, then z_c is set to true if $e_1 < e_2$ and false otherwise. Then new values for the continuous variables are calculated to account for reinit statements and the new values of the discrete variables.

The new values of the discrete and continuous variables may trigger additional events. The algorithm uses the relations in Equations 12–13 and 20–22 to detect these events. This process begins by calculating the values of the zero-crossing functions using $\epsilon = 0$ and the new values of the

discrete and continuous variables. Then for each zero-crossing function generated by a relation, the algorithm checks if either (i) the zero-crossing is negative and z_c is true or (ii) the zero-crossing is positive and z_c is false. For each event generating function, the algorithm checks if at least one of its zero-crossing functions is negative.

If any of these conditions is true, then **z** is reinitialized as before, new values for **q** and **w** are calculated, and the above check for events is repeated. Otherwise, the calculation of the internal transition function is complete with respect to code generated from the Modelica model.

The fourth step is to execute sub-class additions to the internal transition function. This code is supplied with the current values of the Modelica variables, and it may alter the values of Modelica variables and make changes to any variables that are not part of the Modelica model. If sub-class code was executed, then the model's algebraic variables are recalculated and its discrete variables are reinitialized. As before, the values of the discrete variables are checked for consistency with the zero-crossing functions. If inconsistent values are found, then the discrete variables are reinitialized, the continuous variables are recalculated, and this is iterated until a consistent set of values is found.

This procedure for calculating the internal transition function is summarized as Algorithm 2. The bulk of this algorithm, which iterates to find a consistent set of values for **q**, **w**, and **z** following an event, is the same procedure used by OpenModelica's default run-time system (see the *event iteration* algorithm of Braun et al.[23]).

The confluent transition differs from the internal transition function in that the sub-class code is supplied with the input x. In all other respects, a confluent transition is calculated just like an internal transition by using Algorithm 2.

4.4. External transition function

The external transition function determines the response of the model to input from an external source. The arguments to the external transition function are the input x and the

---

ALGORITHM 2: Calculate δ_{int}.

Advance **q** and **w** from t_0 to $t_0 + h$ using $\mathbf{z}(t_0)$
Select new system of equations as needed and recalculate **q** and **w** for this selection
if atEvent = true (see Algorithm 1 and Algorithm 3) **then**
   repeat
      Calculate **z** using **q** and **w**
      Calculate **q** and **w** using **z**
   until z is consistent with its zero-crossing functions
   if there exists sub-class code that has not been executed **then**
      Execute sub-class code using **q**, **w**, and **z**
      goto **repeat**
   end
end

ALGORITHM 3: Calculate δ_{ext}.

---

missedOutput ← {}
Calculate zero-crossing functions for time t_0 and $t_0 + e$
Advance **q** and **w** from t_0 to $t_0 + e$ using $\mathbf{z}(t_0)$
if any zero-crossing function changes sign **then**
    missedOutput ← $\lambda(s)$
    Calculate new state using Algorithm 2 with input x and atEvent = true
else
    Select new system of equations as needed and recalculate **q** and **w** for this selection
    Execute sub-class code using **q**, **w**, **z**, and input x
    repeat
        Calculate **z** using **q** and **w**
        Calculate **q** and **w** using **z**
    until z is consistent with its zero-crossing functions
end

---

time e that has elapsed since the most recent event at time t_0. Upon receiving an input, the part of the model generated by the OpenModelica compiler updates its continuous state variables to time $t_0 + e$. In this respect, the external transition function closely resembles the internal transition function with an addendum to process the input. Input is processed by a sub-class of the model that must be created by the modeler for this purpose.

Unlike the internal transition function, the external transition function must cope with the possibility of an undiscovered state event in the interval $[t_0, t_0 + e]$. This can happen because of the error tolerance p used by Algorithm 1. Recall from Section 4.1.5 that a side of effect of p is to schedule state events some time τ after the model's zero-crossing functions indicate they should occur. This can result in a reordering of internal and external state transitions, with a consequent loss of output from the model. For instance, suppose that an internal event should happen at $t_0 + ta(s)$ but because of p is actually scheduled for $t_0 + ta(s) + \tau$. In this case, an input arriving at $t_0 + ta(s)$ will preempt the output and internal state transition that should have occurred. This disordering of events is due to an unavoidable numerical error in the root finding algorithm.

To test for this condition, the external transition function begins by calculating the zero-crossing functions for time t_0 and $t_0 + e$. There are two possible outcomes.

1. If the signs of the zero-crossing functions are the same at t_0 and $t_0 + e$ then no event was missed and the external transition function advances the continuous variables from t_0 to $t_0 + e$. As with the internal transition function, the model reconsiders its choice of equations and, if a new choice is indicated, recalculates the model's continuous variables to be consistent with this choice. Next, the external transition function invokes its sub-class code to process the input x. If this changes **q**, **w**, or **z**, then the repeat-until loop in Algorithm 2 is used to find consistent values for all of the model's variables.

2. If the signs are not the same, then a state event was missed. In this case, the model omitted an output and an internal event. To remedy this, the external transition function determines what output should have been emitted and stores this result. The new state of the model is then calculated using the same procedure as for the confluent transition function (see Algorithm 2).

In the second case, the stored data forces the time advance function to return zero. As a result, the output function immediately sends the stored data as an output. If it also happens that $\min\{h_1, h_2, h_3\} = 0$, then output that would have occurred naturally as a result of a zero time advance is appended to the stored data. The stored data is discarded at the next state transition. The procedure for calculating the external transition function is summarized as Algorithm 3.

5. A case study: control of a pick-and-place robot

The control of a place-and-pick robot, described by Nedialkov and Ramdani,[30] demonstrates many of the advantages of having Modelica models incorporated into the adevs simulation package. This demonstration extends the example of Nedialkov and Ramdami in two respects.

The first extension is sensors and actuators separated from the control logic by an Ethernet network that is
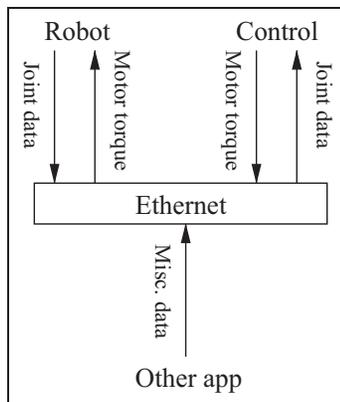
shared among several applications. The media-access control protocol of the network is modeled to capture its effect on packet loss and jitter. Simple models that transmit data packets represent applications on this network other than the robot and its control. The second extension is explicit modeling of the discrete sensing and actuating, which are treated as continuous in the prior work.

The primary components of the model and their interconnections are shown in Figure 3. (The complete source code for the model and simulator are available in the examples directory of the adevs package. This package can be downloaded from http://sourceforge.net/projects/adevs/.) Throughout this modeling problem, we are presented with the option of realizing particular dynamics using either the Modelica language or directly with a

Figure 3. Model comprising a robotic arm, its controller, Ethernet network, and sources of background traffic.

DEVS model. One such choice is periodic sampling of the robot's joint angles. This can be done using the sample function of the Modelica language or with a DEVS model that generates periodic requests for a new sample value.

In most cases, the Modelica language is used just for those elements of the model that have continuous dynamics. Discrete effects that do not depend on continuously evolving variables are implemented with DEVS atomic or coupled models. The exception to this is our use of the Modelica sample function to model fixed rate sensing of the arm angle. This is done to demonstrate the use of time events in a model mixing Modelica and adevs components. Figure 4 shows how the modeling is partitioned between adevs and Modelica, and how the pieces are integrated in the context of Figure 1.

To be consistent with the notation used by Nedialkov and Ramdani,[30] the terms $\mathbf{q}$, z, and others are redefined in the description of the robot, network, and control. However, the model does not refer explicitly to prior uses of these terms, and so their overloading should not cause confusion.

5.1. Robot

The robot consists of two articulated arms that hang from the ceiling and are attached in parallel to a single tool. The relevant parts of the robot are illustrated in Figure 5. The two revolute joints P_1 and P_2, located on the ceiling at x_p and $-x_p$, respectively, are controlled by motors delivering torques Γ_1 and Γ_2. The angles q_1 and q_2 of these joints determine the position x and z of the tool in the x–z plane. The arms are symmetric with two sections of length l and
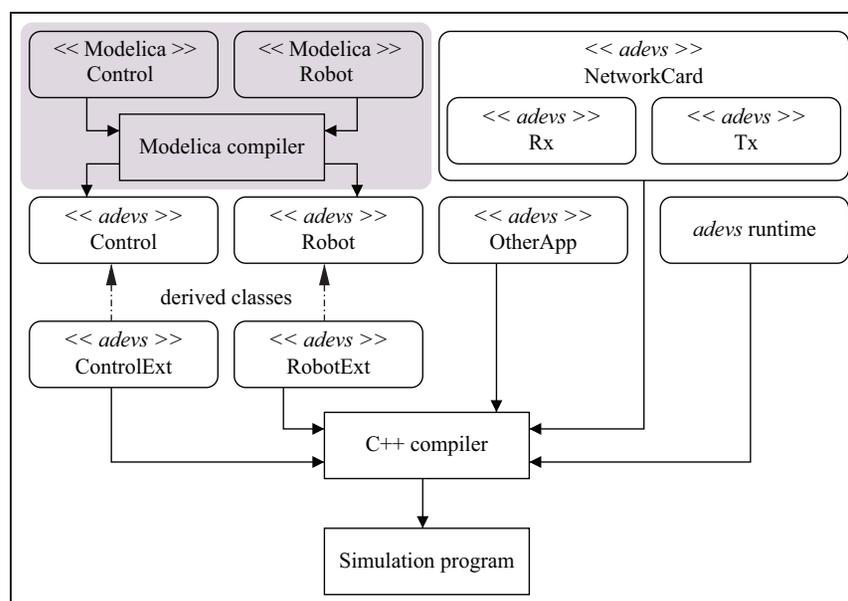
Figure 4. Work flow for the case study and its partitioning into adevs and Modelica components.

L connected by another revolute joint. The position x, z of the tool is related to the angles q_1, q_2 by the algebraic equations

$$0 = (x - x_p - L \cos q_1)^2 + (z + L \sin q_1)^2 - l^2 \quad (26)$$

$$0 = (x + x_p + L \cos q_2)^2 + (z + L \sin q_2)^2 - l^2 \quad (27)$$

Using the notation from Nedialkov and Ramdani,[30] we write $\mathbf{x} = (x, z)$, $\mathbf{q} = (q_1, q_2)$, and Equations 26–27 as $\phi(\mathbf{x}, \mathbf{q}) = 0$.

The torques Γ_1 and Γ_2, denoted collectively by $\boldsymbol{\Gamma} = (\Gamma_1, \Gamma_2)$, on the actuators determine the forces and torques acting on the tool and joints. Using the parameters in Table 1, the equations describing these forces and torques are

$$\boldsymbol{\Gamma} = \boldsymbol{\Gamma}_{red} + \boldsymbol{\Gamma}_{fric} + \boldsymbol{\Gamma}_{arm} + \boldsymbol{\Gamma}_{farm} + \boldsymbol{\Gamma}_{tplate} \quad (28)$$
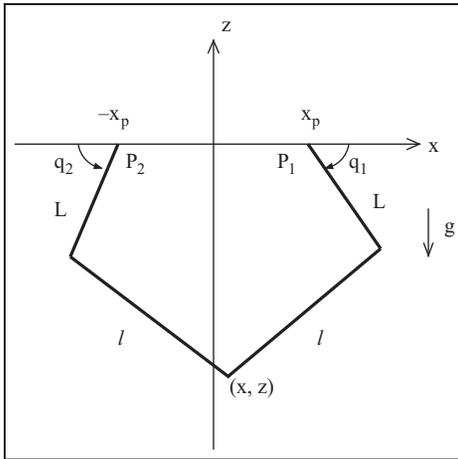
Figure 5. Schematic of the pick-and-place robot.

$$\boldsymbol{\Gamma}_{red} = \nu^2 (J_{mot} + J_{red})\ddot{\mathbf{q}} \quad (29)$$

$$\boldsymbol{\Gamma}_{fric} = \text{sign}(\dot{\mathbf{q}})F_s + F_v \dot{\mathbf{q}} \quad (30)$$

$$\boldsymbol{\Gamma}_{arm} = I\ddot{\mathbf{q}} - m_1 g \cos \mathbf{q} \quad (31)$$

$$\boldsymbol{\Gamma}_{farm} = 0.5 m_2 L (L\ddot{\mathbf{q}} - g \cos \mathbf{q}) \quad (32)$$

$$0 = (\mathbf{J}_x^T \mathbf{J}_q^{-1})\boldsymbol{\Gamma}_{tplate} + (m_2 + m_3 + m_l)(\ddot{\mathbf{x}} + \mathbf{g}) \quad (33)$$

$$\mathbf{J}_q = \partial \phi / \partial \mathbf{q} \quad (34)$$

$$\mathbf{J}_x = \partial \phi / \partial \mathbf{x} \quad (35)$$

$$\mathbf{g} = (0, -g) \quad (36)$$

The sign function is defined such that $\text{sign}(a) = 1$ if $a \geqslant 0$ and -1 otherwise. The functions sign, cos, sin, and differentiation with respect to time apply component-wise to the vector arguments. For the purpose of control, the inputs to this model are the torques $\boldsymbol{\Gamma}$ and the outputs are the joint angles $\mathbf{q}$.

Equations 26–36 are implemented as a Modelica model called BaseRobot. These equations constitute a differential algebraic system with index 3.[30] The OpenModelica compiler reduces Equations 26–36 to family of index 1 systems in the form of Equations 6–7. For a numerically robust simulation, the choice of equations changes dynamically while the simulation executes.[18] This model thereby demonstrates OpenModelica's algorithm for index reduction and the complementary support of the adevs run-time for selecting a system from the resulting family of index 1 DAEs.

The model BaseRobot also includes variables q1_sample and q2_sample that hold sample values for $\mathbf{q}$ and a variable sampleNumber that counts the samples taken. The DEVS output function and internal transition function are extended to emit q1_sample and q2_sample as output whenever sampleNumber changes value.

Table 1. Parameters for the pick-and-place robot.

Description		Notation	Value
Motor position (m)	P_1x-coordinate	x_p	0.1
	P_2x-coordinate	$-x_p$	0.1
Lengths (m)	arm	L	0.3
	forearm	l	0.7
Masses (kg)	arm	m_1	0.82
	forearm	m_2	0.14
	tool (traveling plate)	m_3	0.5
	net load	m_l	5.0
Moment of inertia (kg.m^2)	motor	J_{mot}	0.37×10^{-4}
	gears	J_{red}	9.09×10^{-4}
	arm	J_{red}	0.018895002
Friction coefficients		F_s	3
		F_v	0.5
gear reduction ratio		ν	5
gravity (m/s^2)		g	9.81

Measurements of the joint angles **q** may be generated in two ways. The first method of sampling takes new measurements at a fixed rate. To implement this fixed rate sampling, a new class called Robot1 is derived from BaseRobot. This derived class adds the algorithm section shown below.

```
model Robot1 extends BaseRobot;
  parameter Real sampleFreq = 1000.0;
algorithm
  if initial() or sample(0,1.0/
  sampleFreq) then
    q1_sample := q1;
    q2_sample := q2;
    sampleNumber := sampleNumber + 1;
  end if;
end Robot1;
```

This method of sampling demonstrates support for time events in the extended OpenModelica compiler and new run-time system.

The second method of sampling transmits a new measurement of **q** upon a significant change Δ in x or z. As above, a new class called Robot2 is derived from BaseRobot to implement this method of sampling. This derived class adds the algorithm section shown below.

```
model Robot2 extends BaseRobot;
  parameter Real Delta = 0.0005;
  output Real xsampled, zsampled;
algorithm
  if initial() or abs(x-xsampled) >= Delta or
  abs(z-zsampled) >= Delta then
    q1_sample := q1;
    q2_sample := q2;
    xsampled := x;
    zsampled := z;
    sampleNumber := sampleNumber + 1;
  end if;
end Robot2;
```

This method of sampling demonstrates support for state events in the extended OpenModelica compiler and run-time system.

The Modelica models called Robot1 and Robot2 are compiled to atomic models for adevs which are also called Robot1 and Robot2. These atomic models are then extended by deriving C++ sub-classes named RobotExt1 and RobotExt2 (or, generically, RobotExt). The sub-classes add the capability to act on two types of discrete events.

The first event is a change in the discrete variable sampleNumber. For this purpose, the RobotExt classes have a member variable called sampleNumber that is initially equal to its Modelica counterpart. At any internal or confluent event where these values disagree, the RobotExt sets its time advance to zero. This causes the RobotExt to immediately produce as output the current values of q_1

and q_2. The RobotExt then sets its own sampleNumber variable equal to its Modelica counterpart. The second discrete event is new values for the torques Γ_1 and Γ_2. The RobotExt responds to this input by setting the Modelica variables Γ_1 and Γ_2 to their new values.

A partial listing of the source code for the RobotExt1 is shown in the next section. This listing shows how the internal transition function has been modified to keep track of the sample number and decide when a new sample of the arm angle should be sent through the network. Also shown is how this model sends data via its output function and receives data through is external transition function. The methods set_T that are called in the process_command method adjust the corresponding values in the Modelica model. Likewise, the get_sampleNumber method retrieves the value of the Modelica model's sampleNumber variable and get_q1 and get_q2 return the arm angles.

5.2. Control

Measurements of **q** from the robot are transmitted via the Ethernet network to the control system. Control of the robot is accomplished with separate PID control of q_1 and q_2. The reference trajectory $\mathbf{q}_d = (q_{d,1}, q_{d,2})$ to be followed by each angle is calculated using the geometric constraint ϕ from Equations 26–27 and a reference trajectory $\mathbf{x}_d = (x_d, z_d)$ for the tool.

The control torques are calculated at intervals h_n using a sequence of samples $n = 1, 2, \ldots$ of **q** and the reference trajectories $\mathbf{x}_d$ and $\mathbf{q}_d$. A subscript is appended to each term to indicate which sample value is being considered; e.g., the nth sample of $\mathbf{x}_d$ is indicated by $\mathbf{x}_{d,n}$. For each sample n, the control signal $\mathbf{\Gamma}_n$ is calculated with the system of equations

$$0 = \phi(\mathbf{x}_{d,n}, \mathbf{q}_{d,n}) \tag{37}$$

$$\mathbf{e}_n = \mathbf{q}_{d,n} - \mathbf{q}_n \tag{38}$$

$$\dot{\mathbf{e}}_n = \frac{\mathbf{e}_n - \mathbf{e}_{n-1}}{h_n} \tag{39}$$

$$\int \mathbf{e}_n = \mathbf{e}_{n-1} + h_n \mathbf{e}_n \tag{40}$$

$$\mathbf{\Gamma}_n = 2000 \left(\mathbf{e}_n + \frac{1}{300} \int \mathbf{e}_n + 0.05 \dot{\mathbf{e}}_n \right) \tag{41}$$

To realize this model, the reference trajectory $\mathbf{x}_d$ and Equation 37 are implemented in the Modelica language as a model called Control. From this Modelica model, the OpenModelica compiler generates an atomic model also named Control. This atomic model is sub-classified to create the atomic model ControlExt that realizes Equations 38–41. Although Modelica is clearly a best fit for realizing Equation 37, the choice to implement the other control

```cpp
// Derive a new class from the Robot1 class that was generated by the Modelica compiler.
class RobotExt1: public Robot1{
  public:
    // Input and output ports for the DEVS model
    static const int sample, command;
    // Constructor, destructor, etc.
    .....
    // Internal transition function for the derived class
    void internal_event(double* q, const bool* state_event){
      // Calculate new state variable values for the Modelica model
      Robot::internal_event(q,state_event);
      // Extended model will schedule a time event for the current time if
      // doSample is true (the scheduling code is omitted from this listing).
      doSample = lastSampleNumber != get_sampleNumber();
      lastSampleNumber = get_sampleNumber();
    }
    // External transition function for the derived class. The confluent transition
    // function has been omitted from this list, but is similar to what is shown here.
    void external_event(double* q, double e, const adevs::Bag < OMC_ADEVS_IO_TYPE > & xb){
      // Calculate new state variable values for the Modelica model. The base class does
      // nothing with xb but uses e for the simulation step size.
      Robot::external_event(q,e,xb);
      // Process the incoming data
      process_input_data(xb);
    }
    // Output function for the derived class.
    void output_func(const double *q, const bool* state_event, adevs:: Bag < OMC_ADEVS_IO_TYPE > & yb){
      // Called base class output_func so that the derived class will have access to values in the
      // trial solution at time t + ta(s)
      Robot::output_func(q,state_event,yb);
      // Sample the arm angles and put them into a network packet
      if (doSample){
          SampleSig* sig = new SampleSig(get_q1(),get_q2());
          IO_Type msg;
          msg.port = sample;
          msg.value = new NetworkData(NetworkData::APP_DATA,controlAddr,100,sig);
          yb.insert(msg);
      }
    }
private:
    // Member variable declarations, etc.
    .....
    // Get commands received from the network and send them to the Modelica model
    void process_input_data(const Bag < IO_Type > & xb){
      for (Bag < IO_Type >::const_iterator iter = xb.begin(); iter != xb.end(); iter++ ){
          NetworkData* pkt = dynamic_cast < NetworkData* > ((*iter).value);
          CommandSig* sig = dynamic_cast < CommandSig* > (pkt->getPayload());
          process_command(sig->getT1(),sig->getT2());
      }
    }
    // Send new commands for torque 1 and torque 2 to the motors
    void process_command(double T1, double T2){
      // Set the values of T1 and T2 in the Modelica model
      set_T(T1,0);
      set_T(T2,1);
      // Recalculate the Modelica equations using these new values
      update_vars();
    }
};
```

equations in a derived class is arbitrary. Regardless of this choice, a sub-class of the generated atomic model is necessary for interacting with the network model by which sensor data arrives and control commands are sent.

The model ControlExt operates as follows. The values of $\mathbf{e}_1$, $\int \mathbf{e}_1$, and $\dot{\mathbf{e}}_1$ begin at zero. Upon receiving a new value $\mathbf{q}_n$, $n > 1$, the external transition function of ControlExt calculates Equations 37–41 using for h_n the time elapsed since the previous input. It next sets its time advance to zero and immediately generates as output the new command $\mathbf{\Gamma}_n$. The internal transition function then sets the time advance to infinity, causing the controller to wait for a new measurement.

5.3. Ethernet model

The control is connected to the actuators and sensors of the robot through an Ethernet network that is shared with other applications. These other applications create background traffic that causes dropped packets and jitter in the control loop. For this example, the media access control layer is modeled with adevs, but reuse of existing network models is also feasible; for examples of such reuse see Nutaro et al.[4,5] and Kim et al.[31] The model of media access control presented here is a simplification of the protocol as it is described by Tanenbaum.[32]

The primary elements of the media access control model are the transmit model and receive model called Tx and Rx, respectively. These are components of the NetworkCard model shown in Figure 6. To create a network, the "to_media" output of each NetworkCard is connected to the "from_media" input of every other NetworkCard. An application sends data through the network by generating an input for the "from_app" port of that application's NetworkCard. Data from the network is presented to an application as output from the NetworkCard's "to_app" output port.

Every data packet sent through the network has three fields: (i) the address of the NetworkCard that is to receive the packet; (ii) the payload to be transported to that receiver; and (iii) an indication of the role to be played by a particular instance of that packet. There are four roles: (i) APP_DATA indicating a packet arriving from or going to an application; (ii) TX_START indicating the start of a transmission over the physical Ethernet media; (iii) TX_CMPLT indicating the successful completion of a transmission over the physical Ethernet media; and (iv) TX_FAIL indicating a failed transmission over the physical Ethernet media.

5.3.1. Receiver model. The Rx model has a single input port called "from_media" and single output port called "to_app". The model receives packets with role TX_START, TX_CMPLT, and TX_FAIL on its "from_media" port and generates packets of type APP_DATA on
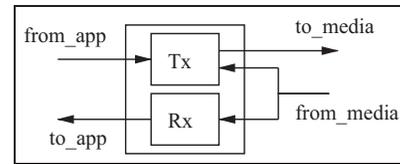
Figure 6. Model of a network card for transmitting and receiving data on the Ethernet network.

its "to_app" output port. The Rx model is initially idle, waiting to receive a packet on its "from_media" port. Upon receiving a packet, it checks the role and address of the packet. If the role is TX_CMPLT and the address matches the address of the NetworkCard, then the Rx model changes the role to APP_DATA and immediately places that packet onto its "to_app" output port. Otherwise the packet is discarded.

5.3.2. Transmitter model. The Tx model has two input ports, called "from_app" and "from_media", and a single output port called "to_media". The Tx model queues packets received on its "from_app" port. Upon receiving such a packet, it is placed at the back of the queue. Queued packets are processed first come, first served.

To process a packet, the Tx model begins by waiting for the physical media to become idle. If the media is not already idle, then the end of the active transmission is indicated by a packet with role TX_FAIL or TX_CMPLT on the "from_media" port. Upon the media becoming idle, the Tx model immediately issues a packet with role TX_START to indicate the start of a transmission over the media.

The time to transmit a packet is expressed in Ethernet frames, each of which is 51.2 μs long and contains 512 bits. For a packet with b bytes, the transmission time is $\lceil 8b/512 \rceil \times 51.2$ μs. If this time elapses without interruption, i.e., without receiving any input on the "from_media" port, then the model indicates a successful transmission by sending the packet with role TX_CMPLT on the "to_media" output port. This indicates to all other network cards that the transmission is completed.

Otherwise, the transmission is interrupted and the Tx model increments a counter-indicating the number of times that it has attempted to send the packet. Then the Tx model waits for a time equal to a number of Ethernet frames selected at random from 0 to the value of the counter or 10, whichever is smaller. When this time has elapsed, the transmitter waits for the media to become idle before issuing a packet with role TX_START on the "to_media" port.

The above process is repeated until either the transmission is completed without interruption or the attempt to transmit has failed 16 times. At the 16th failure, the Tx model issues a packet with role TX_FAIL on the

"to_media" port to indicate it has stopped the attempted transmission. The packet is then discarded and work on the next packet, if any, begins.

5.4. Performance of the control as a function of background load

The control trajectory (x_d, z_d) for this example is

$$x_d(t) = -0.35 \sin(\pi t/2.0) \tag{42}$$

$$z_d(t) = -0.7 + 0.1 \cos(\pi t/2.0) \tag{43}$$

The initial position of the tool is $(x, z) = (0, -0.6)$ with $\Gamma_1 = \Gamma_2 = 0$. The ability of the control to direct the arm depends on both the method of sampling and the quantity of background traffic on the network.

The maximum data rate for the Ethernet network is approximately 512 bits per 51.2 μs, which is 1,250,000 bytes per second. Each packet generated by the angle sensor and torque control contains 100 bytes. With a fixed sampling rate of 1000 Hz, the sensor and control consume approximately 16% of the network's capacity. For the given control trajectory, the threshold sensor generates data at slightly lower rates, which vary from about 300 Hz up to about 1000 Hz, thereby also occupying about 16% of the network's capacity.

The remainder of the traffic is created by background sources. Each background source generates 1% of the maximum data rate of the network by transmitting 125 packets per second on average, with each packet having 100 bytes. The actual interval between transmissions is sampled from an exponentially distributed random variable with mean of 1/125 seconds.

To transmit these packets, each background source is connected to the network through its NetworkCard model. Notably, the time for a successful transmission of 100 bytes is $\lceil (100 \times 8)/512 \rceil \times 51.2 = 819\,\mu$s. The actual transmission delay is therefore quite small and control errors due to the network are caused by the protocol for access to the network media.

Table 2 shows the maximum norm of the control error $|x - x_d| + |z - z_d|$ over 20 s of operation. This table shows results for 0, 10, 20, 30, 40, and 50 sources of background traffic. Unsurprisingly, the control degrades as the network becomes congested. Nonetheless, the control is stable with as much as 30% background loading. Adding the approximately 16% load due to the control, this demonstrates stable control at a total network utilization of up to 46%. For the cases with 50 sources (for sampled control) and 40 sources (for quantized control), the control is unstable and the simulation is stopped prior to 20 s.

5.5. Advantages relative to co-simulation

The advantages of the proposed approach are apparent when compared with its alternatives, which are distinguished by their lack of the time advance function. Without the time advance (or its equivalent) the simulation algorithm cannot precisely coordinate the exchange of data between components. In this case, it becomes necessary to select synchronization points $t_1, t_2, \ldots$ at which the components are stopped and their outputs exchanged; see, e.g., the Modelica technical report "Functional Mock-up Interface for Co-Simulation,"[33] which describes this approach to co-simulation. The explicit selection of synchronization points requires a choice between the accuracy of the simulation and the speed of its execution.[8]

If we use this approach to co-simulation, then to achieve a simulation of the robot, network, and control with timing errors smaller than an Ethernet frame requires a time step $t_{k+1} - t_k < 51.2\,\mu$s. For the 20 s simulation, this small time step causes approximately 400,000 steps of the co-simulation algorithm. However, the different components in the model interact over milliseconds, not microseconds. If we increase the time step to 1 ms, then the simulation needs only 20,000 steps of the co-simulation algorithm, but every transaction is artificially delayed by the duration of 20 Ethernet frames.

In contrast to this, discrete event simulation of the Modelica model achieves better than microsecond accuracy in the timing of interactions and millisecond leaps in time between these interactions. Indeed, there are just 20,000 interactions between the discrete event and continuous models during a simulation of the robot and its control. The Modelica model of the robot generates 20,000 discrete outputs and receives 20,000 discrete inputs.

Moreover, the discrete event simulation is more accurate than the co-simulation with a microsecond time step. The timing of the network events is simulated with machine precision, and the detection of state events is accurate to within the detection threshold p, which is 10^{-7} in this example, i.e., fractions of a micrometer for the arm

Table 2. Maximum norm of the control error observed over 20 seconds.

Number of sources	Error, sampled control	Error, quantized control
0	0.0037	0.0037
10	0.0049	0.0037
20	0.0053	0.011
30	0.0091	0.013
40	0.037	> 1.6
50	> 2.3	> 2.1

position and fractions of a microradian for the arm angle. Hence, the interaction between the method of sampling and the protocol for media access control is simulated more precisely than in a co-simulation using microsecond time steps, and this precision is achieved with less computation.

6. Conclusion

The OpenModelica extensions described here, and those described in prior work[34–36] on precisely integrating continuous and discrete event models, embed the numerical solver for the continuous model within a discrete event simulation. The precise management of time in these discrete event simulations depends on the capability of components in the model to provide the time of their next internal event. With this information, the event schedule of the simulator synchronizes, naturally and precisely, the exchange of data between every component of the combined model. The time advance functions of the continuous models are what distinguishes this approach from its alternatives.

Nonetheless, the proposed approach to combined simulations is compatible with other approaches to co-simulation. This compatibility is based in large part on the DEVS formalism and an extensive body of research describing its use within federated simulation environments. Indeed, the structure of the discrete event simulation facilitates its integration with other co-simulation tools. This has been discussed in detail with regard to DEVS and the HLA,[37–40] and similar approaches have been used in other co-simulation frameworks.[31,41]

In principle the approach described here for DEVS-based simulation packages can be adapted for event-oriented simulation packages. This is described conceptually in a prior article on the split-system method,[4] and partial implementations of this concept appear in much earlier work.[36] If such an adaptation can be accomplished, it would extend the capability to simulate very sophisticated, hybrid models to the majority of simulation packages for discrete event models. Conversely, the proposed approach can extend packages for simulating continuous models to encompass very sophisticated, hybrid dynamic systems.

It is noteworthy that if the challenges described in Section 2 for QSS integrators and the dummy derivatives method could be resolved, then it would be possible to achieve the benefits of the proposed approach *and* obtain the significant reductions in execution time afforded by the QSS methods. Future research in this direction is particularly attractive because the split system method cannot take advantage of some optimizations that are routinely done in the simulation loop for a standalone solver. One example of this is in the calculation of the time advance

function where we must choose between a hobbled root finding algorithm or increasing the memory required for the simulation. It is likely that these undesirable choices could be eliminated if the functions for F and G in the split system method were realizable using QSS type numerical methods.

References

1. Carloni LP, Passerone R, Pinto A, et al. Languages and tools for hybrid systems design. *Found Trends Electron Design Automat* 2006; 1(1/2): 1–193.
2. Lee EA and Zheng H. Operational semantics of hybrid systems. In Morari M and Thiele L (eds.), *Hybrid Systems: Computation and Control* (*Lecture Notes in Computer Science*, vol. 3414). Berlin: Springer, 2005, pp. 25–53.
3. Zeigler BP, Praehofer H and Kim TG. *Theory of Modeling and Simulation,* 2nd edn. New York: Academic Press, 2000.
4. Nutaro J, Kuruganti PT, Protopopescu V, et al. The split system approach to managing time in simulations of hybrid systems having continuous and discrete event components. *Simulation* 2012; 88(3): 281–298.
5. Nutaro J. *Building Software for Simulation: Theory and Algorithms, with Applications in C++* . New York: Wiley, 2010.
6. Nutaro J, Kuruganti P, Miller L, et al. Integrated hybrid-simulation of electric power and communications systems. In *IEEE Power Engineering Society General Meeting*, pp. 1–8.
7. Sydney A, Nutaro J, Scoglio C, et al. Simulative comparison of multiprotocol label switching and openflow network technologies for transmission operations. *IEEE Transactions on Smart Grid* 2013; 4(2): 763–770.
8. Hopkinson K, Wang X, Giovanini R, et al. EPOCHS: a platform for agent-based electric power and communication simulation built from commercial off-the-shelf components. *IEEE Transactions on Power Systems* 2006; 21(2): 548–558.
9. Broman D, Brooks C, Greenberg L, et al. *Determinate composition of FMUs for co-simulation*. Technical Report UCB/EECS-2013-153, University of California at Berkeley, Electrical Engineering and Computer Sciences, 2013. http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-153.pdf.
10. Fritzson P, Aronsson P, Lundvall H, et al. The OpenModelica modeling, simulation, and development environment. In *Proceedings of the 46th Conference on Simulation and Modeling (SIMS'05)*, pp. 83–90.

11. Fritzson P. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. New York: Wiley-IEEE Press, 2014.

12. Sanz V, Urquia A, Cellier FE, et al. System modeling using the Parallel DEVS formalism and the Modelica language. *Sim Modell Practice Theory* 2010; 18(7): 998–1018.

13. Beltrame T. *Design and Development of a Dymola/Modelica Library for Discrete Event-oriented Systems using DEVS Methodology*. Master's Thesis, ETH Zürich, Department of Computer Science, Institute of Computational Science, 2006.

14. D'Abreu M and Wainer G. M/CD++ : modeling continuous systems using Modelica and DEVS. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 229–236.

15. Floros X, Bergero F, Cellier F, et al. Automated simulation of Modelica models with QSS methods - the discontinuous case. In *8th International Modelica Conference*, pp. 657–667.

16. Floros X, Cellier F and Kofman E. Discretizing time or states? a comparative study between DASSL and QSS. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pp. 107–115.

17. Bergero F, Floros X, Fernández J, et al. Simulating Modelica models with a stand-alone quantized state system solver. In *Proceedings of the 9th International Modelica Conference*, pp. 237–246.

18. Mattsson SE and Söderland G. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing* 1993; 14(3): 677–692.

19. Mattsson SE, Olsson H and Elmqvist H. Dynamic selection of states in Dymola. In *Modelica Workshop* 2000, pp. 61–67.

20. Nutaro J and Zeigler B. On the stability and performance of discrete event methods for simulating continuous systems. *J Computat Phys* 2007; 227(1): 797–819.

21. Brenan K, Campbell S and Petzold L. *Numerical solution of initial-value problems in differential-algebraic equations*. Philadelphia, PA: SIAM, 1996.

22. Hindmarsh A, Brown P, Grant K, et al. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM Trans Math Softw* 2005; 31(3): 363–396.

23. Braun W, Bachmann B, Pro S, et al. Synchronous events in the OpenModelica compiler with a Petri net library application. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pp. 63–70.

24. Lundvall H, Fritzson P and Bachmann B. *Event handling in the OpenModelica compiler and runtime system*. Technical Report 2, Linköping University, Department of Computer and Information Science, PELAB - Programming Environment Laboratory, 2008.

25. Frenkel J, Kunze G and Fritzson P. Survey of appropriate matching algorithms for large scale systems of differential algebraic equations. In *Proceedings of the 9th International Modelica Conference*, pp. 433–442.

26. Ralston A and Rabinowitz P. *A first course in numerical analysis, second edition*. New York: McGraw-Hill, 1978.

27. Zeigler BP. *Theory of Modeling and Simulation*. New York: John Wiley & Sons, 1976.

28. Zhang J, Johansson KH, Lygeros J, et al. Zeno hybrid systems. *International Journal of Robust and Nonlinear Control* 2001; 11(5): 435–451.

29. Pulecchi T and Casella F. HyAuLib: modelling Hybrid Automata in Modelica. In *Proceedings of the 6th International Modelica Conference*, volume 1, pp. 239–246.

30. Nedialkov NS and Ramdani N. *Towards Integrating Hybrid DAEs with a High-Index DAE Solver*. Rapport de recherche RR-6834, INRIA, 2009. http://hal.inria.fr/inria-00360999.

31. Kim T, Hwang MH, Kim D, et al. DEVS/NS-2 environment: integrated tool for efficient networks modeling and simulation. In *Proceedings of the 2007 spring simulation multiconference, SpringSim'07*, volume 2, pp. 219–226.

32. Tanenbaum AS. *Computer Networks,* 3rd edn. Englewood Cliffs, NJ: Prentice-Hall, 1996.

33. Modelisar. *Functional Mock-up Interface for Co-Simulation, Version 1*. Technical Report 07006, ITEA 2, 2010.

34. Kofman E. Discrete event simulation of hybrid systems. *SIAM J Sci Comput* 2004; 25(5): 1771–1797.

35. Lee EA and Zheng H. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proceedings of the 7th ACM and IEEE international conference on Embedded software (EMSOFT'07)*, pp. 114–123.

36. Klingener J. Combined discrete-continuous simulation models in ProModel for Windows. In *Proceedings of the 1995 Winter Simulation Conference*, pp. 445–450.

37. Zeigler BP, Hall SB and Sarjoughian HS. Exploiting HLA and DEVS to promote interoperability and reuse in Lockheed's corporate environment. *Simulation* 1999; 73(5): 288–295.

38. Sarjoughian H and Zeigler B. DEVS and HLA: complementary paradigms for modeling and simulation? *Trans Soc Comput Sim Int* 2000; 17(4): 187–197.

39. Kim YJ and Kim TG. A heterogeneous simulation framework based on the DEVS BUS and the high level architecture. In *Proceedings of the Winter Simulation Conference*, volume 1, pp. 421–428.

40. IEEE. IEEE Standard for Modeling and Simulation (M & S) High Level Architecture (HLA) - Framework and Rules. *IEEE Standard 1516-2000*, 2000; i–22.

41. Huang D, Sarjoughian H, Wang W, et al. Simulation of semiconductor manufacturing supply-chain systems with DEVS, MPC, and KIB. *IEEE Trans Semicond Manuf* 2009; 22(1): 164–174.