

PDEVS-BASED HYBRID SYSTEM SIMULATION TOOLBOX FOR MATLAB

Christina Deatcu
Birger Freymann
Thorsten Pawletta

Research Group Computational Engineering and Automation
University of Applied Sciences Wismar
Wismar, D-23966, Germany
{christina.deatcu, birger.freymann, thorsten.pawletta}@hs-wismar.de

ABSTRACT

MATLAB/Simulink is a popular software environment used by engineers and scientists. It offers integrated modeling and simulation tools and numerous toolboxes employable in conjunction with modeling and simulation tasks. However, modeling and simulation of discrete event systems is often confusing and sometimes not compliant with the established system theory. This contribution presents a MATLAB toolbox for hybrid system modeling and simulation, based on an extended Parallel Discrete Event System Specification (PDEVS) formalism, called hybrid PDEVS. The formalism is introduced, focusing on the integration of MATLAB's built-in solvers for differential equations. Additionally, specific features regarding the modeling and simulation process and debugging support are discussed. Finally, the combination with other MATLAB methods and toolboxes is illustrated by means of two examples. The toolbox should contribute to bringing the PDEVS formalism into the engineering community and to support research in system simulation in conjunction with other numerical methods.

Keywords: PDEVS; Hybrid Modeling; Debugging; MATLAB/Simulink®, Hybrid System Simulation.

1 INTRODUCTION

For many years, modeling and simulation (M&S) has been used extensively in the engineering community, not just to study complex system's behavior but also in close conjunction with applications, such as for commissioning and operation of control applications, and with other numerical methods and tools. Beside the separated fields of continuous and discrete event M&S, models of hybrid systems are a big challenge for engineers, in particular. In this context, hybrid means that the system dynamic is characterized by a combination of discrete events and continuous processes. In the field of simulation research, there is always an interest in investigating the integration of simulation methods with other novel computational methods, as currently for machine learning and data analysis (Lawrence Livermore National Laboratory 2016).

Scientific and Technical Computing Environments (SCEs), such as the widely used MATLAB/Simulink environment, provide a large number of classical and recent computational methods for use in a single uniform environment interactively. These also include various tools for M&S. In the graphical Simulink environment, hybrid models can be built and executed using various blocksets and tools. However, Simulink itself was originally designed for continuous system models. Probably, this is a reason that the modeling and the simulation process of discrete event oriented system dynamics is sometimes confusing and not compliant with the established theory. Currently, a new discrete event framework has been introduced that draws more inspiration from system theory according to (Li, Mani, and Mosterman 2016).

However, our own experiments with the first two releases (R2016a, R2016b) delivered different results for models with event cascades. Moreover, the debugging process showed that the execution seems not to be really event oriented. The nondisclosure of the underlying simulation algorithms is a particular problem for using the tools in simulation research and simulation education and results sometimes in a time-consuming error search when developing applications. The presented MATLAB toolbox, called MatlabDEVS, implements the Parallel Discrete Event System Specification Formalism's (PDEVS) model definitions and simulation algorithms following (Zeigler, Kim, and Prähofer 2000). Users familiar with this formalism will have the chance to model and simulate discrete event systems compliant with the simulation theory. Moreover, they can combine the simulation method with all other computational methods provided by the MATLAB/Simulink environment.

The most natural way for modeling and simulating hybrid systems based on PDEVS is the Quantized State Systems (QSS) approach (Zeigler and Lee 1998, Kofman and Junco 2001). The MatlabDEVS toolbox also provides some examples for using QSS. However, the QSS approach is hardly known by engineers. Therefore, a method to combine the PDEVS simulation algorithms with MATLAB's ODE solvers has been introduced (Pawletta et al. 2006). This combination with advanced ODE solvers as provided by MATLAB is unique among PDEVS tools and has been further improved with the new release. Furthermore, an interface to allow easy integration of external hardware as well as of arbitrary MATLAB methods and toolboxes into a PDEVS based simulation is available. Using this feature, for instance complex image data processing tasks or machine learning algorithms can be intuitively combined with PDEVS simulation by engineers. Co-simulations involving other MATLAB simulators can be executed in a straightforward manner. Most other tools, like for example *adevs* (Nutaro 2017) or *VLE* (Virtual Laboratory Environment 2017; Quesnel, Duboz, and Ramat 2009), either require manual programming based on C++ or other languages to integrate with advanced methods or they use standards like FMI (Functional Mock-up Interface) for co-simulation implementation, which is much more labor-intensive.

The paper starts with summarizing the theoretical basics of the hybrid PDEVS formalism. Then, the toolbox's software structure and basics regarding the modeling and simulation process are presented. After that, some novel features offered by the toolbox with version 2.0 (FG CEA 2016), such as error detection during modeling, debugging assistance during simulation and post-mortem debugging after a simulation run, are introduced and discussed. Finally, we present two example models, to illustrate how the toolbox can be combined with other MATLAB methods and tools. The main objective of the paper is to give an impression how well known theoretical M&S methods are adapted to bring it in operation within an accepted numerical software tool.

2 HYBRID PDEVS FORMALISM

The PDEVS formalism has its origin in the classic DEVS formalism. It was introduced by (Chow 1996) and is under continuous development, for example in the extension by ports in (Zeigler, Kim, and Prähofer 2000) and their improvement in (Schwatinski and Pawletta 2010). Generally, both formalisms are based on a modular, hierarchical modeling and simulation approach. Regarding the model specification they distinguish between two types of models. *Atomic* models are used for specifying dynamic system behavior, while *coupled* models are used for specifying system aggregations. Analogous, the formalisms introduce two types of processors for model execution, called *simulator* and *coordinator*. As mentioned in the previous section, an objective of the toolbox is to combine the PDEVS formalism with ODEs to describe continuous system behavior and to integrate time discretization methods for numerical solving of ODEs. Subsequently, the essential extensions that have been introduced are described.

2.1 Formal Model Definitions

Prähofer defined an extension of classic DEVS for using ODEs to specify continuous system behavior, and added the explicit Euler method to the DEVS simulator to allow computation of *hybrid atomic DEVS*

(Prähofer 1992). Regarding the formal definition of atomic models, we have adapted Prähofer's idea to atomic PDEVS and call the new model type *hybrid atomic PDEVS*, which is defined as follows:

$$A_{hybridp} = (X, Y, S, f, c_{se}, \lambda_c, \delta_{state}, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda_d, ta)$$

The sets of inputs X , outputs Y and states S of hybrid atomic PDEVS may contain discrete as well as continuous values. The continuous dynamics is mapped by the rate of change function f and the continuous output function λ_c . Events occurring in such a hybrid atomic model can be external or internal events or they can be *state events*, which are caused by discontinuities of continuous state values. The state event condition function c_{se} defines the conditions under which such state events occur. In contrast to Prähofer's definition, we specify state transitions caused by state events separated from state transitions caused by external events, using the two transition functions δ_{state} and δ_{ext} instead of one common $\delta_{x\&s}$ function. As usual in PDEVS, internal events cause output events that are external events for other models. Output events are specified in the discrete output function λ_d . Subsequently a state transition, that is defined by the internal transition function δ_{int} , is executed. The confluent function δ_{conf} specifies the state transitions in case of simultaneous external and internal events. Often, modelers use the confluent function just to specify in which order the other transition functions should be executed but, of course, any desired behavior can be specified there. Finally, the time advance function ta specifies the time point of the next internal event after each event based state transition.

For coupled models we use the standard definition of PDEVS, which is as follows:

$$N_{hybridp} = (X_N, Y_N, D, \{M_d \mid d \in D\}, EIC, EOC, IC)$$

However, note that the sets of inputs X_N and outputs Y_N can contain discrete as well as continuous values. The set D specifies the subcomponents, which may be atomic or coupled PDEVS models. Subcomponents can be either pure discrete or hybrid models. The set M_d is the corresponding index set. Coupling relations are described by the three sets: (i) *EIC* for external input couplings; (ii) *EOC* for external output couplings and (iii) *IC* for internal couplings. In contrast to standard PDEVS, a coupling relation can be discrete or continuous and we use ports to structure the coupling relations according to (Schwatinski and Pawletta 2010). Couplings from one output port to multiple input ports are possible and, additionally, links between several output ports to one input port can be specified.

2.2 Basic Execution Algorithms and New Messages

For executing a modular hierarchical PDEVS model, each atomic model is linked with a *simulator* and each coupled model with a *coordinator*, which are generally called *processors*. The model execution is driven by passing messages between these processors. We use the four improved messages for PDEVS with ports, introduced in (Schwatinski and Pawletta 2010): (i) initialization message (i, t) ; (ii) internal state transition message $(*, t)$; (iii) input message (x, t) and (iv) output message (y, t) . The overall simulation loop is controlled as usual by a third type of processor, called the *root coordinator*. The model execution starts by sending an initialization message (i, t) to all processors recursively. Now the root coordinator knows the next event time t_{next} . Next, in a pure discrete event based PDEVS model, the root coordinator sends a $(*, t)$ message with $t=t_{next}$ to the processor linked with the topmost model component. Now all events at t are handled by message passing between the processors and the t_{next} value of each model is calculated. Afterwards, the root coordinator starts the next cycle by sending a $(*, t)$ message, where t is the minimal t_{next} value, to the processor linked with the topmost model component.

In a hybrid model with ODEs, specifying a continuous system behavior, between the consecutive event times $t_{next_1}, t_{next_2}, \dots$ the continuous state transitions have to be computed. As has already been mentioned, Prähofer extended the processors of the classic DEVS formalism to solve ODEs using the explicit Euler method. However, his approach cannot be extended in a straight forward manner to support advanced time discretization methods for solving ODEs, such as implicit integration algorithms, predictor corrector

methods or methods with adaptive step size control. Therefore, we have developed another approach based on a *wrapper* method as an interface between the PDEVS processors and the advanced numerical ODE solvers provided by the SCE MATLAB. Using the wrapper method, a MATLAB ODE solver is called to compute continuous state transitions in the interval between two consecutive event times. However, the computation of continuous model parts with MATLAB's ODE solvers requires a closed representation of the entire continuous model specification, which may be distributed over the modular, hierarchical model. That means, one vector containing all continuous variables, one function to calculate all derivatives and one function to specify all state events caused by continuous variables is needed. Therefore, we introduce three new messages for the PDEVS processors, which manage the additional information that are necessary for the continuous system specification, as follows:

- (z, τ, \dots) *message* to create/update internal data structures for continuous state, input and output variables and state event conditions
- $(z2, \tau, \dots)$ *message* to establish/update links between continuous outputs and inputs
- (se, τ, \dots) *message* to handle current state events detected by the ODE solver.

Additionally, the root coordinator has to be extended for executing hybrid models. This topic will be discussed in more detail in the next section.

3 THE MATLABDEVS TOOLBOX

The MatlabDEVS toolbox has been under development for about 20 years. Starting with a classic DEVS implementation, it evolved into PDEVS variants and extensions for structure variable, real time and hybrid systems. We aim to bring the general ideas of simulation theory closer to engineers by offering tools embedded in their most common SCE. The toolbox is not open source but it is available for free (FG CEA 2016). It is mostly used for educational purposes and in research projects. This section describes the basic software structure, modeling process and simulation process.

3.1 Basic Software Structure

Current version 2.0 (FG CEA 2016) comes with a lean graphical user interface, offering simulation of several example models and an entry to the HTML-documentation. The modeling process and simulation startup is based on programming with the MATLAB language.

From the beginning, the toolbox was implemented taking advantage of MATLAB's object oriented programming features. All classes used in the toolbox are derived from the MATLAB handle class, so that instances of a class can be called by reference. As depicted in Figure 1, the link between models and their processors is realized by class derivation. Therefore, atomic model specifications always have to be derived from the base classes `atomic` or `hybridatomic`. Coupled model specifications can be defined analogously by deriving from the base classes `coupled` or `hybridcoupled`. However, they can also be instantiated directly from base classes and initialized using predefined methods. Only classes with the prefix `hybrid` support the extensions for hybrid PDEVS as introduced in the previous section. A modular hierarchical model can include pure discrete as well as hybrid models. Of course, a coupled model needs to be a `hybridcoupled`, if at least one hybrid subcomponent exists in its sub-hierarchy.

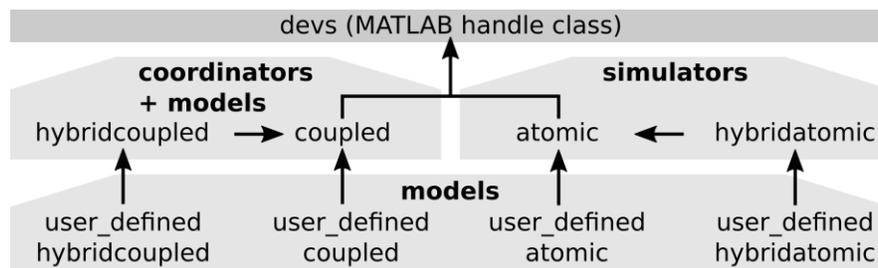


Figure 1: Class hierarchy of MatlabDEVS toolbox.

Besides the classes for modeling and simulation, the toolbox comes with two different *root coordinators* for pure discrete or hybrid models to control the overall simulation. Additionally, several helper functions, e.g. for checking model consistency before simulation, are provided.

3.2 Modeling Process

Modeling needs to be done bottom up, starting with the specification of atomic models. These have to be defined by deriving new user-defined classes from the `atomic` or `hybridatomic` base class. To create one's own atomic models one can use the provided template files, which illustrate how to define x and y ports, discrete and continuous states, system parameters and the characteristic functions. All available MATLAB tools, built in methods and additional toolboxes can be used to specify the behavior of atomic models. Coupled models can be defined in a similar fashion, by deriving user defined classes from the `coupled` or `hybridcoupled` base class. Template files are provided too. In this case, the constructor method of user defined coupled models needs to instantiate all sub-models and to define ports and couplings. Coupled models defined in that way are reusable and can be stored in a model base just as atomic models. Alternatively, non-reusable coupled models can be directly defined within a script file by creating instances of base class `coupled` or `hybridcoupled` and adding ports, sub-components and couplings using base class methods. Couplings are defined via coupling matrixes that define a coupling relation consisting of two pairs (component name, port name) in each row.

Finally, for defining an overall model, the atomic and coupled models that are part of it need to be instantiated in an *initialization script*. The toolbox provides several examples of initialization scripts with different levels of complexity. A minimal initialization script just incarnates the overall model. More complex scripts call further MATLAB methods for data pre- or post-processing, hardware integration or the like.

3.3 Simulation Process

The result of the modeling process is an instance of class `coupled` or `hybridcoupled`, hereinafter called `root_model`, which encapsulates the overall modular, hierarchical model with the associated processors. For simulation, the root model is passed to a *root coordinator*. There are two different root coordinators available. Pure discrete event based models can be computed with the *discrete root coordinator*, which implements the standard PDEVS simulation algorithm, as depicted in Listing 1. The call at the MATLAB prompt or from within a MATLAB script or function is as follows:

```
root_model = r_c_discrete(root_model,tstart,tend)
```

After initialization of the modular, hierarchical model with initial states and computation of each model's next internal event time, recursive $(*, t)$ messages are sent until the simulation time is over or a stop condition is fulfilled. Stop conditions can be set in any part of the model via a global variable. After simulation is over, the total state of the modular, hierarchical model is returned in the `root_model`.

Hybrid models have to be simulated using the *hybrid root coordinator*, which is defined as follows:

```
[root_model,tout,yout,teout,yeout,ieout]=r_c_hybrid(root_model,tstart,tend,plot_params)
```

Besides the `root_model` itself, the hybrid root coordinator returns matrices containing times and function values of continuous variables in `tout` and `yout`. Additionally, times at which state events occur are returned in `teout` and corresponding state values in `yeout` and indices of associated event functions that occur are returned in `ieout`. These values are obtained from return values of the attached MATLAB ODE-solver.

Hybrid PDEVS simulation is divided into three phases as depicted in Listing 2. After the initialization, it is determined if the next phase needs to be a discrete or a continuous one, depending on the next internal discrete event time `tnext`.

```

begin discrete root coordinator

set current simulation time to start time

# initialize the model
tnext <- send (i,t) message to outermost coupled

# simulation loop
while simulation time not over && no stop condition
  # carry out internal event(s) and
  # determine time of next internal event(s)
  tnext <- send (*,t) message to outermost coupled
end while

end discrete root coordinator

```

Listing 1: Basic PDEVS root coordinator.

```

begin hybrid root coordinator

set current simulation time to start time
set phase to 0 # start the initialization phase

while simulation time not over && no stop condition

  switch phase
  -----
  case 0 # initialization phase
  # determine time of first internal event
  tnext <- send (i,t) message to root_model

  if tnext == current simulation time
    set phase to 1 # start a discrete phase
  elseif tnext > current sim. time
    set phase to 2 # start a continuous phase
  end

  -----

  case 1 # discrete phase
  # carry out internal event(s) and
  # determine time of next internal event(s)
  tnext <- send (*,t) message to root_model

  if tnext == current sim. time
    set phase to 1 # remain in discrete phase
  elseif tnext > current sim. time
    set phase to 2 # start a continuous phase
  end

  -----

  case 2 # continuous phase
  # establish references to atomic sim. objects and
  # cont. state variables in vectors aSimObj and cSc
  [aSimObj, cSc] <- send (z,t,..) message to root_model

  # establish direct links between outputs and inputs
  send (z2,t,..) message to root_model

  sort vector aSimObj depending on moore/mealy behavior

  # specify ODE-wrapper with flag=0 as event function
  configure ODE-solver with continuous model

  # call ODE-solver with ODE-wrapper, flag=1,
  # as model function
  # integration interval:
  # [current sim. time, tnext or tfinal]
  [times,values,events] <-
    ODE-solver(ODE-wrapper(1,aSimObj,cSc),interval)

  # set current sim. time to biggest value in times
  current sim. time <- max(times)

  if ODE-solver has returned state events
    # send state-event-message to
    # localize atomic obj. where event has occurred.
    # simulator of this atomic calls its deltastate func.
    # determine time of next internal event
    tnext <- send (se,t,..) message to root_model
  end

  if tnext == current simulation time
    set phase to 1 # start a discrete phase
  else
    set phase to 2 # remain in continuous phase
  end

  -----

end switch
end while
end hybrid root coordinator

```

Listing 2: PDEVS root coordinator for hybrid models.

In a discrete phase, where the current simulation time $t=t_{next}$, a $(*,t)$ message is sent to carry out internal event(s), analogous to basic PDEVS. A continuous phase is entered, if the current simulation time t is smaller than t_{next} . For the interval $[t, t_{next}]$ the continuous model parts have to be computed. That means that continuous state transitions must be calculated and potential state events need to be detected. To provide the necessary continuous model representation required for MATLAB's ODE solver, a continuous phase starts with the newly introduced (z, t, \dots) and (z_2, t, \dots) messages. The return values `aSimObj`, `cSc` contain the continuous model information. This data is used to configure the `ode_wrapper` method, which provides the interface for the MATLAB ODE solver. Thereby, the `ode_wrapper` method acts as event function and as derivation function either, depending on its first input parameter, a flag that is set to 0 or 1.

For a non-structure variable system the entire `ode_wrapper` configuration can be done one-time in the initialization phase. However, the toolbox also supports structure variable systems, in which continuous model parts can change during simulation, as discussed in (Pawletta et al. 2006). The computation overhead is negligible. After the additional continuous data model is built, MATLAB's ODE-solver is called and runs until the end of the current continuous cycle (t_{next}) or until it is interrupted by the occurrence of a continuous state event. The current simulation time t is determined from the return value `times`. If any state events have occurred, an `(se, t, ...)` message is sent to the `root_model` and transmitted to the processors of the modular, hierarchical model to evoke the state event transition function δ_{state} in the affected hybrid atomic models. Additionally, the next internal event times are recalculated and the smallest t_{next} value is returned. Based on a comparison of t_{next} and the current simulation time t , the next phase is determined.

4 ERROR DETECTION AND DEBUGGING SUPPORT

MatlabDEVS provides no graphical support for modeling and only a few visualization features for the simulation process to keep it lean but it is open to MATLAB. Therefore, users can implement features manually easily. However, the toolbox offers some support for debugging and is designed in a way that some of the most frequently occurring modeling errors are prevented. These features are described in this section. An overview on theoretical foundations of debugging DEVS models and a comparison of features included in other popular DEVS tools can be found in (Van Mierlo, Van Tendeloo, and Vanghelouwe 2016). How these features are supported by the toolbox is summarized in Table 1.

Table 1: Debugging criteria following (Van Mierlo, Van Tendeloo, and Vangheluwe 2016).

Criteria	Sup. by MatlabDEVS	Criteria	Sup. by MatlabDEVS
Pause	Manually	State changes	No
(Scaled) real time	Manually	State visualization	Yes/Manually
Big step	Manually	Event visualization	Manually
Small step	Yes	Tracing	Yes
Termination condition	Yes	Model visualization	Manually
Breakpoints	Manually	Reset	Manually
Event injection	Manually	Step back	No (Manually)

4.1 Before Simulation

To avoid the error of trying to set states that are not defined for a specific atomic model class, all set access trials on states are monitored when an atomic model is instantiated and executed. For coupled models, a structure check is available. After instantiation of the root model, its reference can be passed to the `check()` function to find out, if all coupling relations defined in the sets EOC, EIC and IC are valid. It is checked recursively to see if all subcomponents exist, if all connected port names are valid, if all defined ports are used and if there are no self-loops. Furthermore, the base classes offer display functions for all atomic and coupled model types. Depending on the model type, they allow input and output ports, states, subcomponents, the coupling relations, times of last and next event, and information to be shown if debugging and observation of state variables is activated.

4.2 During Simulation

The toolbox offers textual and visual information during a simulation run. This information can be used for debugging. There are three debug modes available. The textual debug modes 1 and 2 provide information

about simulation messages interchanged between associated simulators and coordinators, calls of the dynamic functions of atomic models and current input, output and state values. Moreover, these modes are well suited for learning PDEVs and for understanding the simulation algorithms. For both modes the following can be defined – whether the textual output should be written to: (i) standard output device (the MATLAB prompt); (ii) standard error device or (iii) a text file. Debug mode 2 generates exactly the same output as mode 1 but it operates stepwise. A step is, thereby, not defined as an equidistant amount of time but as the distance between two internal events. The debug mode 3 provides an online visual representation of ports and states and their current values. Figure 2 depicts a detail of two atomics as they are represented in debug mode 3. All state changes are visualized (small steps). The bar at the top contains the name of the atomic model and the name of the coupled model it is part of. The first atomic model displayed in Figure 2 is named ‘am_m2’ and is a subcomponent of ‘cm_c3’. It has two input ports ‘in1’ and ‘in2’, four state variables and one output port ‘out1’.

Each of the three debug modes can be set for the entire model, the `root_model` or for parts of it, by calling the `set_debug(obj,mode,out)` method and passing the object reference, the debug mode and an optional parameter `out` to define output, if the mode is set to 1 or 2, such as follows:

```
set_debug(root_model,1,1) for entire model, mode=1, out= stdout
set_debug(...cm_c3,2,2) for coupled and it's submodels, mode=2, out= stderr
set_debug(...am_m2,3) for single atomic, mode=3
```

The setting is done in the previously discussed initialization script. Furthermore, a modeler can add any kind of visualization that is active during simulation manually, e.g. an ODE-plot as it is used in the hybrid example later on.

4.3 After Simulation

Debugging after a simulation run is over, the so-called post-mortem debugging, is supported by the textual debug modes if one decides to write the output to a file. Moreover, state variables in atomics can be automatically recorded as (time, value) pairs using the `set_observe(obj,mode)` method. Setting the observe mode works similarly to debugging and can be set for the entire model or for selected parts of the modular, hierarchical model. Continuous states of hybrid models are recorded by default and are part of the return values of the hybrid root coordinator. Hence, the states are available for any kind of analysis, prints and processing with any standard or additional MATLAB tools after simulation.

5 EXAMPLES

To illustrate how models can be designed and simulated using the toolbox and taking advantage of the integration with MATLAB, we discuss extracts of two examples. The first one demonstrates how MATLAB’s well established ODE solvers are used within a PDEVs simulator for computing continuous system dynamics. The second example is intended to demonstrate the integration of real hardware and other MATLAB toolboxes.

5.1 Hybrid System Simulation

The problem is taken from the book (Pedgen, Sadowski, and Shannon 1995). It is a reference example for hybrid system simulation. It considers the processes taking place in a bottling plant, consisting of the arrival of trucks with juice, the temporary storage of juice in a tank and the canning process itself. The truck arrivals and the docking and undocking processes of trucks at the tank are described by probability distributions. The juice pumping to the canning machine and the canning process are modelled using ODEs.. Additionally, there are several operational rules for the processes:

1. If the tank for temporary storage with its capacity of 6000 gallons is filled, suspend pumping juice from the truck
2. If the filling level of the tank has sunk to 5500 gallons again, restart pumping

3. If the tank for temporary storage is empty (5 gallons left), stop filling the containers
4. If a new truck arrives and pumping to the tank is started, the filling of containers is resumed

The problem was modeled as a `hybridcoupled` PDEVS, consisting of five pure discrete event `atomic` models - `t_gen` (generates trucks), `queue` (waiting trucks), `dock/undock` (connect/disconnect truck to tank), `transducer` (for statistics) – and three `hybridatomic` models – `truck` (pumping process), `tank` (temporary storage), `canner` (filling process). Each of the `hybridatomic` models include two continuous variables, one for the level value and one for the rate of change value, such as `TankLevel` and `TankRate`. The operation rules defined are expressed by the state event condition functions c_{se} of the `hybridatomic` models and by triggering events as reactions to conditions that have occurred. Code 1 shows the appropriate implementation part for the `hybridatomic tank`. If, for example, operation rule 1 is fulfilled, the `hybridatomic tank` triggers a state event by evaluating the event number in its δ_{state} function, which triggers a simultaneous internal event. Hence, after executing the time advance function ta with $\sigma=0, \lambda_d$ and δ_{int} are executed, which send a ‘stop_filling’ event to the `hybridatomic truck` and schedule the next internal event at $t=\infty$ for `tank`. Continuous state visualization during simulation as depicted in Figure 3 is readily achieved by using the MATLAB’s ODE-plot functionality, which can be configured in the initialization script. A description of the whole model is part of the toolbox documentation (FG CEA 2016).

cm_c3 : am_m2					
Port	Value	State	Value	Port	Value
in1	{ }	phase	'passive'	out1	{ }
in2	{[1]}	sigma	Inf		
		q1	0		
		q2	1		

cm_g : am_g1					
Port	Value	State	Value	Port	Value
		counter	3	p1	{[1]}

Figure 2: Example of a visual output in debug mode 3.

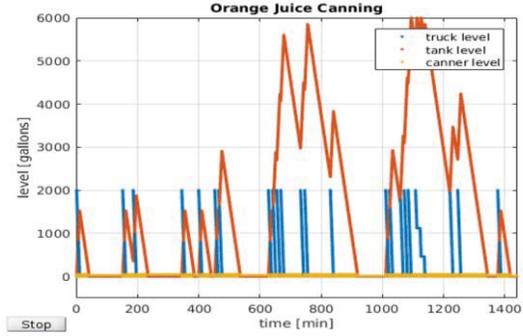


Figure 3: Example of an ODE-plot for hybridatomics.

5.2 Discrete Event Oriented Robot Control

This example highlights the fundamental idea of using a PDEVS model, implemented with MatlabDEVS, as the control software based on the Software in the Loop (SIL) method. However, this real world engineering example is too complex for describing it in detail here. In the following, we will only outline the fundamentals of the applied theoretical approach. We then focus our description on the integration of real hardware and external software components to demonstrate the benefits of integrating a PDEVS simulation environment with an engineering software tool such as MATLAB. A detailed discussion of the theoretical approach based on PDEVS can be found in (Schwatinski et al. 2010). Implementation details of the other external software components used are described in (Deatcu et al., 2015, FG CEA 2011).

There exist several extensions for DEVS and PDEVS for connecting with external hardware and software under real-time conditions, such as RT-DEVS (Cho and Kim 2001) and PDEVS-RCP (Schwatinski et al. 2010). We prefer atomic PDEVS-RCP models as the interface because the necessary extensions are part of the model specification. Hence, they can be executed with the ordinary PDEVS algorithms. An atomic PDEVS-RCP model is extended by a set of executable activities $a \in S_A$, which are part of the state set, defined as $S_{RCP} = S \cup S_A$. Additionally, it contains a set of time values $t_{real} \in X_{clock}$ from a real-time clock (RTC), which is part of the input set, defined as $X_{RCP} = X \cup X_{clock}$. The RTC is a special PDEVS-RCP without an input set. An activity a is a tuple $a = (m, p_1, \dots, p_n, r, t_{min}, t_{max})$, consisting of a method named m , input arguments p_1, \dots, p_n , return value r and execution time boundaries $[t_{min}, t_{max}]$.

```

function ret = cse(obj,gt,y)
%           |           |           |
%           |           |           | integration termination event (1/0)
%           |           |           | zero-crossing direction from + to -
%           |           |           | (1 if event fct increases, -1 if decreases)
%           |           |           |
ret = [6000-y(1),1,-1;...% ev. #1, if TankLevel reaches 6000 from down
      5500-y(1),1, 1;...% ev. #2, if TankLevel reaches 5500 from up again
      5-y(1),1,1]; % ev. #3, if TankLevel is 5
end

function obj = deltastatefun(obj,gt,y,event_number)
if event_number == 1 % if tank filled up to 6000
    obj.s.event_type = 'stop_filling';% filling can be continued
    obj.s.filling_suspended = 1; % to remember, that tank was full
    obj.s.sigma = 0; % trigger internal event and output before
end
if event_number == 2 % if TankLevel sunk to 5500
    ...
end
if event_number == 3 % if tank is empty
    ...
end
end

function ta = tafun(obj)
    ta = obj.s.sigma; % sigma = 0 after state evt., inf after int. evt.
end

function lambdafun(obj)
if strcmp(obj.s.event_type,'stop_filling')
    obj.y.out1 = {0}; % send truck a don't-fill-signal
elseif strcmp(obj.s.event_type,'continue_filling')
    obj.y.out1 = {1}; % send truck a do-fill-signal
else
    % if strcmp(obj.s.event_type,'stop_canning')
    obj.y.out2 = {0}; % send the canning a tank-empty message
end

function deltaintfun(obj)
    obj.s.sigma = inf;
end

```

Code 1: State event detection and reactions through the example of the hybridatomic tank, coded in MatlabDEVS.

Additionally, the discrete output function λ_d is extended to a combined output and activity mapping function, defined as $\lambda_d: S_{RCP} \rightarrow Y \times S_A$. Of course, an atomic PDEVS-RCP can specify ordinary PDEVS behavior and coupled PDEVS models can contain ordinary PDEVS and atomic PDEVS-RCP models respectively. Because the communication with external components is encapsulated in the activities, a control model can be stepwise developed from a pure simulation model to an SIL control system. The activities can be defined using all methods and tools provided by the MATLAB environment. Subsequently, we focus on this part though our example.

Figure 4 shows the basic structure of such a PDEVS based control application using different actuators and sensors. An input buffer (IB) contains parts of different types at fixed positions, which have to be sorted by a robot (R) into two output buffers (OB₁, OB₂). For sorting, the R has to pick a part from IB and move it to a scale (S) and then to a camera (C). Depending on both identification results, the weight and the color, R moves the part to fixed positions at OB₁ or OB₂. The components **R**, **C**, **S** and the real-time clock **RTC** are implemented as atomic PDEVS-RCP models and the buffers as atomic PDEVS models. For communicating with the external components, the atomic PDEVS-RCP models define activities. Activities can use methods

provided by the MATLAB/Simulink environment or third party toolboxes, such as the *Robot Control and Visualization Toolbox* (RCV) for MATLAB (Deatcu et al. 2015, FG CEA 2011). Some of the methods are delineated in Figure 5. The time boundaries of the activities are very different because some are based on synchronous and others on asynchronous communication.

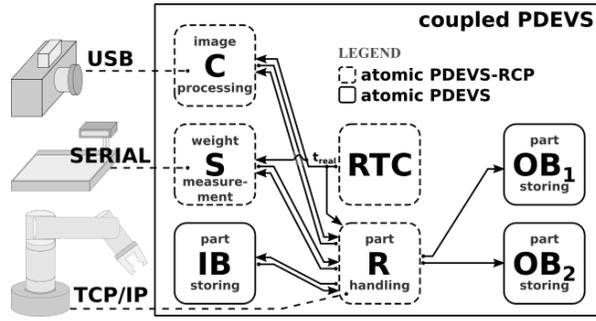


Figure 4: Basic structure of the PDEVS based control application.

activity methods of C from IMAGE Processing Toolbox

```
cam = webcam();           %connect to webcam
img = snapshot(cam);      %take a picture
delete(cam);              %disconnect webcam
```

activity methods of S from Instrumental Control Toolbox

```
com = serial(SerialPort,'BaudRate',Value)
fopen(com);                %open SerialPort
fprintf(com,CommandString) %write to SerialPort
fclose(com)                 %close SerialPort
```

activity methods of R from RCV Toolbox

```
tra = load('trajectory.mat'); %load points
rob = robot('open',RobotType,'tcpip',IP_Adr,PORT); %con. to robot
pos = rget(rob,'pose');      %get current position
id = rmove(rpoint(x,y,t,o,a,t)); %move to new position
tf = ris(id);                 %is motion complete?
robot('close',rob);          %disconnect
```

Figure 5: Activity methods for the application in Fig. 4.

6 CONCLUSION

The PDEVS Hybrid System Simulation Toolbox for MATLAB, named MatlabDEVS, offers the usage of the well-defined PDEVS modeling and simulation formalism in the widely used MATLAB/Simulink environment. It can be combined with other MATLAB methods and tools. Hence, the toolbox may contribute to bringing the PDEVS formalism into the engineering community, where it could be particularly interesting for research and education purposes. A typical application field could be the development of recent discrete event controls, such as that exemplified by a small robotic control application. The availability of PDEVS within MATLAB may also support research on system simulation in conjunction with other computational methods, such as for machine learning or big data problems. In our group, a recent research topic, taking into account the MatlabDEVS toolbox, is the development of metamodeling techniques and automated model transformations in the simulation domain (Pawletta et al. 2016).

REFERENCES

- Cho, S. M., and T. G. Kim. 2001. "Real Time Simulation Framework for RT-DEVS Models". *Transactions of The Society for Computer Simulation International*, vol. 18 no. 4, pp. 203–215.
- Chow, A. C.-H. 1996. "Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and its Distributed Simulators". *Transactions of The Society for Computer Simulation International*, vol. 13 no. 2, pp. 55–67.
- Deatcu, C., B. Freymann, A. Schmidt, and T. Pawletta. 2015. "MATLAB/Simulink Based Rapid Control Prototyping for Multivendor Robot Applications". *SNE – Simulation Notes Europe*, vol. 25 no. 2, pp. 69–78. ARGESIM/ASIM Pub. TU Vienna, Austria.
- FG CEA 2016. "The MatlabDEVS Website". https://www.mb.hs-wismar.de/cea/DEVS_Tbx/MatlabDEVS_Tbx.html. Accessed Sept. 14, 2016.
- FG CEA 2011. "Robotic Control & Visualization Toolbox for Matlab Website". https://www.mb.hs-wismar.de/cea/sw_projects.html. Accessed Sept. 14, 2016.
- Kofman, E., and S. Junco. 2001. "Quantized-State Systems: a DEVS Approach for Continuous System Simulation". *Transactions of The Society for Computer Simulation International*, vol. 18, pp. 123–132.

- Lawrence Livermore National Laboratory 2016. “A Converging Path for Simulation, Machine Learning and Big Data”. <https://computation.llnl.gov/newsroom/simulation-machine-learning-big-data-converging-path>. Accessed Oct. 12, 2016.
- Li, W., R. Mani, and P. J. Mosterman. 2016. “Extensible Discrete-Event Simulation Framework in SimEvents”. In *Proceedings of the 2016 Winter Simulation Conference*, 12 pages (pre-print).
- Nutaro, J. . ”A Discrete Event System Simulator”. <http://web.ornl.gov/~1qn/adevs/index.html>. Accessed Feb. 24, 2017.
- Pawletta, T., C. Deatcu, O. Hagendorf, S. Pawletta, and G. Colquhoun. 2006. “DEVS-Based Modeling and Simulation in Scientific and Technical Computing Environments”. *Proceedings of the 2006 Spring Simulation Conference – TMS/DEVS*, pp. 151–158.
- Pawletta, T., A. Schmidt, B. P. Zeigler, and U. Durak. 2016. “Extended Variability Modeling Using System Entity Structure Ontology within MATLAB/Simulink”. *Proceedings of the 2016 Spring Simulation Conference – ANSS*, pp. 62–69.
- Pedgen, C. D., R. P. Sadowski, and R. E. Shannon. 1995. *Introduction to Simulation Using SIMAN*. 2nd ed. New York, NY, USA, McGraw-Hill.
- Praehofer, H. 1992. *System Theoretic Foundations for Combined Discrete-Continuous System Simulation*. Ph.D. thesis. VWGÖ, Vienna, Austria.
- Quesnel, G., R. Duboz, and É. Ramat. 2009. “The Virtual Laboratory Environment - An Operational Framework for Multi-Modelling, Simulation and Analysis of Complex Systems”. *Simulation Modelling Practice and Theory*, vol. 17, pp. 641-653.
- Schwatinski, T., and T. Pawletta. 2010. “An Advanced Simulation Approach for Parallel DEVS with Ports”. *Proceedings of the 2010 Spring Simulation Conference – TMS/DEVS*, pp. 132–139.
- Schwatinski T., T. Pawletta, S. Pawletta, and C. Kaiser. 2010. “Simulation-based Development and Operation of Controls on the Basis of the DEVS Formalism”. *Proceedings of the 7th EUROSIM Congress*. vol. 2, 8 pages.
- Van Mierlo, S, Y. Van Tendeloo, and H. Vangheluwe. 2016. “Debugging Parallel DEVS”. *Transactions of The Society for Computer Simulation International*, published online before print. Available via <http://msdl.cs.mcgill.ca/people/yentl/papers/2016-DebuggingPDEVS.pdf>. Accessed Sept. 14, 2016.
- Virtual Laboratory Environment. “VLE Project Website”. <http://www.vle-project.org/>, Accessed Feb. 24, 2017.
- Zeigler, B. P., T. G. Kim, and H. Prähofer. 2000. *Theory of Modeling and Simulation*. 2nd ed. San Diego, CA, USA, Academic Press.
- Zeigler, B. P., and J. Lee. 1998. “Theory of Quantized Systems: Formal Basis for DEVS/HLA Distributed Simulation Environment”. In *SPIE proceedings*, pp. 49–58.

AUTHOR BIOGRAPHIES

CHRISTINA DEATCU is a research engineer at Wismar University, Germany. Her research interests lie in theory of modeling and simulation, especially of discrete event and hybrid systems. Her email address is christina.deatcu@hs-wismar.de.

BIRGER FREYMANN holds a Master's degree for Mechanical Engineering and is currently working on his PhD thesis at the field of robot control design. His email address is birger.frey mann@hs-wismar.de.

THORSTEN PAWLETTA is a Professor for Applied Computer Science at Wismar University, Germany. His research interests lie in M&S theory and application in engineering. His email address is thorsten.pawletta@hs-wismar.de.