Co-simulation of Distributed Embedded Real-Time Control Systems[*]

Marcel Verhoef[1], Peter Visser[2], Jozef Hooman[3], and Jan Broenink[2]

[1] Chess, P.O. Box 5021, 2000 CA Haarlem and Radboud University Nijmegen, Institute of Computing and Information Sciences, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
Marcel.Verhoef@chess.nl

[2] University of Twente, Control Engineering, Department of Electrical Engineering, Mathematics and Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands
P.M.Visser@utwente.nl, J.F.Broenink@utwente.nl

[3] Embedded Systems Institute, P.O. Box 513, 5600 MB Eindhoven and Radboud University Nijmegen, Institute of Computing and Information Sciences
hooman@cs.ru.nl

Abstract. Development of computerized embedded control systems is difficult because it brings together systems theory, electrical engineering and computer science. The engineering and analysis approaches advocated by these disciplines are fundamentally different which complicates reasoning about e.g. performance at the system level. We propose a lightweight approach that alleviates this problem to some extent. An existing formal semantic framework for discrete event models is extended to allow for consistent co-simulation of continuous time models from within this framework. It enables integrated models that can be checked by simulation in addition to the verification and validation techniques already offered by each discipline individually. The level of confidence in the design can now be raised in the very early stages of the system design life-cycle instead of postponing system-level design issues until the integration and test phase is reached. We demonstrate the extended semantic framework by co-simulation of VDM++ and bond-graph models on a case study, the level control of a water tank.

Keywords: simulation, continuous time, discrete event, VDM++, bond graphs.

1 Introduction

Computers that are intimately coupled to the environment which they monitor and control are commonly referred to as embedded systems. We focus on the class

[*] This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project was partially supported by the Dutch Ministry of Economic Affairs under the Senter TS program.

of embedded systems that control a physical process in the real world. We refer to these systems as embedded control systems. Examples are the control unit of a washing machine and the fuel injection system in a private car. Embedded control systems execute an algorithm that ensures the correct behavior of the system as a whole. The common element of all these systems is that timeliness is of concern. Control actions have to be taken *on time* to keep the physical process in the required state. I.e., embedded control systems are real-time systems.

This is in particular true for the class of high-tech systems such as for instance wafer steppers and high-volume printers and copiers. The productivity of these machines, which is often their most important selling point, depends on the performance of the embedded control system. Typically, these complex machines are composed of several subsystems that need to work together to get the job done, which may require multi-layer and distributed control. For example, each subsystem may have its own embedded control system to perform its specific function while another, dedicated, subsystem coordinates the system as a whole by telling the other subsystems what to do and when. It is not hard to imagine that the design of the control strategy for these systems is challenging.

This is complicated by the fact that systems are often developed out-of-phase. Typically, mechanical design precedes electronics design which precedes software design. Although there is a trend towards concurrent engineering to reduce development time, the lead times for mechanical design and engineering typically still exceed those of electronics and software. System level design considerations are validated during the test and integration phase, which may cause significant delays in the project if an important issue was overlooked. Software is often the only part of the system that can be changed at this late stage. These late changes can cause a significant increase in the complexity of the software, especially when a carefully designed software architecture is violated to compensate for some unforeseen problems in the hardware. Hence, it is important to get as much feedback as possible in the earliest stages of the system design life-cycle, to prevent this situation.

Model-based design addresses this challenge. Reasoning about system-level properties is enabled by creating abstract, high-level and multidisciplinary models of the system under construction. Mono-disciplinary models typically allow optimization of single aspects of the design, while multidisciplinary models allow reasoning about fitness for purpose across multiple system aspects. Suppose, for instance, that the position of a sheet of paper in the paper path of a printer is measured with a sensor that generates an interrupt when the edge of the sheet is observed. High interrupt loads can occur on the embedded control system if these sensors are placed physically close together, because they are triggered right after one another. A very powerful processor may be required in order to deal with this sudden peak load, in particular when a short response time must be guaranteed for each event. There is a clear trade-off between spatial layout and performance in this example. Analysis of multidisciplinary models provides valuable insight into the design such that these trade-offs can be made in a structured way, earlier, and with more confidence.

This approach was studied in the Boderc project [1] in which the authors participated. We observed that creating multidisciplinary models is far from trivial. The notations and the engineering and analysis approaches that are advocated by the involved disciplines are different and the resulting models are typically not at the same level of abstraction. Henzinger and Sifakis [2] even claim that these are fundamental problems and that a new mathematical foundation is required to reason about these integrated multidisciplinary models. The approach taken in this paper is different. We would like to be able to combine the state of the art in each discipline in a useful and consistent way. In other words, we want to construct multidisciplinary models from mono-disciplinary models. We are certainly not the first to propose this idea but we believe that our solution to this problem is novel.

Contribution of this paper. We have reconciled the semantics of two existing formal notations such that system models, which are composed of sub-models written in either language, can be conveniently studied in combination. We also demonstrate how this is achieved in practice by tool coupling. The result is a light-weight modeling approach that enables construction of multidisciplinary models that can be simulated, in addition to the analysis techniques already available for each sub-model individually. Moreover, the reconciled semantics ensures reliable simulation results which can be obtained with little effort.

Structure of this paper. An overview of the current state of practice is presented in Section 2. Modeling and analysis of embedded control systems is discussed by introducing a motivating case study in Section 3. The results of the simulation using the tool coupling are shown in Section 4. The semantic integration is presented from a formal perspective in Section 5. Finally, we look at related and future work and we draw conclusions in Section 6.

2 Current State of Practice in Academia and Industry

The importance of model-based design is widely recognized and we observe that many contenders, typically originating from a specific discipline, are extending their techniques to cater for this wider audience. Matlab/Simulink is an example of this trend. In combination with their Stateflow and Real-time Workshop add-on products, they provide a tool chain for embedded systems design and engineering. It is particularly well-suited for fine grained controller design. This is not surprising because the roots of the tools are firmly based in systems theory. Stateflow can be used to model the control software using finite state machines. However, this technique is not very convenient for specifying complex algorithms. One has to write so-called S-functions or provide a piece of C-code in order to execute the Stateflow model. Timing is idealized by the assumption that all transitions take a fixed amount of timer ticks. Scheduling and deployment of software on a distributed system cannot easily be described and analyzed. Henriksson [3] designed and implemented the TrueTime toolkit on top of Simulink which provides a solution for describing scheduling and deployment, but the software models remain at a low abstraction level. We believe that these tools are *not* acceptable to the embedded software engineer at large, because insufficient support is provided for modern software engineering approaches to design and implement complex real-time software.

A similar situation arises from IBM Rational Technical Developer (formerly known as Rational Rose Real-time) and I-Logix Rhapsody. These software development environments are increasingly used in real-time embedded systems development [4]. They provide modeling capabilities based on the Unified Modeling Language (UML) and the System Modeling Language (SysML) and are supported by mature development processes (RUP and Harmony respectively). Both tools aim to develop executable models that are deployed on the target system as soon as possible to close the design loop. This requires the model to evolve to a low level of abstraction early in the design process in order to achieve that goal. Actions are coded directly in the target (programming) language and timing can be specified by using so-called timer objects provided by the modeling framework. However, their resolution and accuracy is determined by the services of the operating system running on the target platform, they are not part of the modeling language. Moving code from one platform to another might lead to completely different timing behavior. Similarly, task priorities and scheduling are implementation specific. We believe that these tools are *not* acceptable to the control engineer at large, because no support is provided to design and analyze the control laws that the system should implement.

Is it possible to support control and software engineers using a single method or tool? Several attempts have been made to unify both worlds. For example, Hooman, Mulyar and Posta [5] have co-simulated Rose Real-time software models with control laws specified in Matlab/Simulink. They removed the platform dependent notion of time in Rose Real-time by providing a platform neutral notion of time instead. This is achieved by development of an interface that sits in between Rose Real-time and Simulink, which exposes the software simulator of Rose Real-time to the Simulink internal clock. While this is a step forward, it also shows that Rose Real-time is not very suitable for the co-simulation of control systems, because it lacks a suitable notion of simulation time and the run-to-completion semantics does not allow interrupts due to relevant events of the physical system under control. I-Logix has recently announced integration of Rhapsody with Simulink but the technical details have not yet been unveiled.

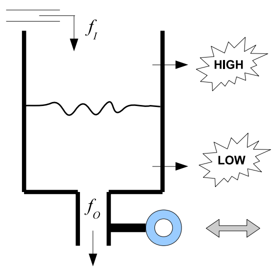
Lee et al [6] propose a component based, actor oriented approach. They define a framework in which all components are concurrent and interact by sending messages according to some communication protocol. The communication protocol and the concurrency policies together are called the model of computation. Ptolemy-II [6] is a system-level design environment that supports heterogeneous modeling and design using this approach. It supports several domains, each of which is based on a particular model of computation, such as for example discrete event, synchronous data flow, process networks, finite state machines and communicating sequential processes. They can be combined at liberty to describe the system under investigation. This approach seems to be a major step forward for model based design of real-time embedded systems, but paradoxically, it does not appeal to either control engineers or software engineers. Perhaps the approach proposed by Ptolemy-II upsets the current way of working so much that it is considered too high a risk to use in an industrial environment. Currently, only simulation is offered as a means of model validation and synthesis is under development for some domains. Verification of Ptolemy-II models is not yet possible because the semantics of actors has not been formally defined.

3 Modeling and Analysis of Embedded Control Systems

The complexity of embedded control design and analysis is probably best explained by means of a motivating example. We use the level control of a water tank in this paper. This example is small and simple, but it contains all the basic elements of an embedded control system. These elements are presented in detail in this section. An overview of the case study is presented in Figure 1. The case study concerns a water tank that is filled by a constant input flow f_{I} and can be emptied by opening a valve resulting in an output flow f_{O}. The volume change is described by equations (1) and (2), where A is the surface area of the tank bottom, V is the volume, g is the gravitation constant, ρ is the density of liquid and R is the resistance of the valve exit.

From the system theoretic point of view, we distinguish the *plant* and the *controller* of an embedded control system, as shown in Fig. 2. The plant is the physical entity in the real world that is observed and actuated by the controller. More accurately, we study feedback control in this paper. Feedback controllers compute and generate a control action that keeps the difference between the observed plant state and its desired value, the so-called set-point, within a certain allowed margin of error at all times. The plant is a dynamic system that is usually described by differential equations if in the continuous time (CT) domain or by difference equations if it is described in the discrete time (DT) domain.

The water tank case study is an example of a continuous time system, described by differential equation (1). Controllers observe some property of the plant and they change the state of the plant by performing a control action, according to some control law. This control law keeps the system as a whole in some desired state. In our case study, the water level is observed by three



$$\frac{d\mathrm{V}}{dt} = f_{\mathrm{I}} - f_{\mathrm{O}} \tag{1}$$

$$f_{\mathrm{O}} = \begin{cases} \frac{\rho \cdot g}{\mathrm{A} \cdot \mathrm{R}} \cdot \mathrm{V} & \text{if valve = open} \\ 0 & \text{if valve = closed} \end{cases} \tag{2}$$

Fig. 1. The water tank level control case study

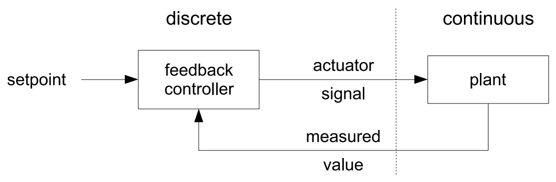
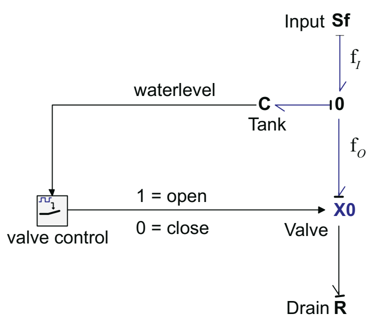


Fig. 2. System theoretic view of a control system

sensors: a pressure sensor at the bottom of the tank which measures the current water level continuously and two discrete sensors that raise an alarm if a certain situation occurs. The top sensor informs us when the water level exceeds the high water mark and the bottom sensor fires if the water level drops below the low water mark. The aim of the controller is to keep the water level between the low and high watermark. The controller can influence the water level by opening or closing a valve at the bottom of the tank. We assume that the valve is either fully open or fully closed. Plant modeling and controller descriptions are discussed in more detail in the following sections.

3.1 Plant Modeling

For modeling the plant of the embedded control system, we use so-called bond graphs [7,8] in this paper. Bond graphs are directed graphs, showing the relevant dynamic behavior of the system. Vertices are the sub-models and the edges, which are called *bonds*, denote the ideal (or idealized) exchange of energy. Entry points of the sub-models are the so-called *ports*. The exchange of energy through a port (p) is always described by two implicit variables, effort ($p.e$) and flow ($p.f$). The product of these variables is the amount of energy that flows through the port. For each physical domain, such a pair of variables can be specified, for example: voltage and current, force and velocity. The half arrow on the vertex at the bonds shows the positive direction of the flow of energy, and the perpendicular stroke indicates the computational direction of the two variables involved. They connect the energy flows to the two variables of the bond. The equations that define the relationship between the variables are specified as real equalities, not as assignments. Port variables obtain a computational direction (one as input, the other as output) by means of computational causal analysis on the graph. This efficient algorithm ensures that the underlying set of differential equations can be solved deterministically by rewriting the equations as assignment statements such that a consistent evaluation order is enforced whenever a solution is calculated. Bond graphs are physical-domain independent, due to analogies between the different domains on the level of physics. Mechanical, electrical, hydraulic and other system parts can all be modeled with bond graphs. Bond graphs may be mixed with block diagrams in a natural way to cover the information domain. Control laws are usually specified with block diagrams and the plant is specified with bond graphs to model a controlled mechatronic



```
variables
  real volume, level;
parameters
  real area = 1.0;
  real gravity = 9.81;
  real density = 1.0;
equations
// p.e = pressure, p.f = flow rate
// integrate flow to obtain volume
  volume = int(p.f);
  level = volume / area;
  p.e = gravity * level * density;
```

Fig. 3. The bond graph plant model of the water tank case study

system. Figure 3 shows the bond graph plant model of the water tank case study. The S_f element is the input flow f_{I}. The C element describes the water tank, the equations are next to the figure. The R element describes the drain. The X0 element is a so-called switching junction which describes the valve. When the valve is opened, a flow f_{O} will be drained from C. There is no flow from C when the valve is closed.

Differential equations are the general format for representing dynamic systems mathematically. For specifying a plant model many continuous-time representations exist, e.g., bond graph models, ideal physical models, block and flow diagrams and so on. A common property is that all these model types are directly related to a set of differential equations. For the subset of linear time-invariant plant models, alternative description techniques exist, such as the s-plane, frequency response and state-space formats [9].

System theory has provided many analysis techniques for time-invariant linear models and design techniques for their associated controllers, for which certain properties can be proven to hold. However, real world systems often tend to be nonlinear and time varying. The task of the control engineer is to find a suitable linearization such that system theory can still be applied to design a controller. Alternatively, simulation can be used if the dynamic system can be described by a collection of so-called ordinary differential equations. This includes the linear time-invariant models mentioned earlier, as well as non-linear and time varying differential equations. Partial differential equations can be approximated by lumped parameter models in ordinary differential equations and also non-deterministic (or stochastic) models can be simulated. Although simulation can never provide hard answers, it is often used because it can address a much larger class of problems than linear analysis. For example, it can be used to determine whether a linearized model is a good abstraction of the original non-linear model, since both models can be simulated.

The basic method used in simulation is to solve a differential equation numerically instead of analytically. Approximations of the solution are computed by means of integration of the differential equations. These numerical integration techniques are commonly referred to as "solvers" and they exist in many flavors. Examples of well-known solvers are Euler, Runge-Kutta and Adams-Bashforth [10,11]. These solvers belong to the class of fixed step size integration algorithms. Also many variable step size algorithms exist and selection of the right solver is non-trivial and requires a good understanding of the model itself. For example, variable step size solvers are typically required when the dynamic system is described by (combined CT and) DT models. In addition, since an approximation of the solution is computed, an integration error is introduced. This error might lead to instability if the solver, and its parameters, are not carefully selected.

3.2 Controller Description

According to Cassandras and Lafortune [12], a system belongs to the class of discrete event systems if the state can be described by a set of discrete values and state transitions are observed at discrete points in time. We adopt this definition here. Discrete event models can be used to describe the behavior of digital computers, which implement certain control laws. Computers execute instructions based on a discrete clock. The result of an instruction becomes available after a certain number of clock ticks has elapsed. Sensor input samples and actuator output values are seen as discrete events in this model of computation.

In order to bridge the gap between continuous time and discrete event simulation, we obviously need to introduce the notion of events in the continuous time solver. Here, we distinguish two different event types: a) state events and b) time events. State events occur when the solution of a differential equation reaches some value p. Time events occur when the solver has reached some time t. Consider a solver that produces a sequence of time steps *time* and a sequence of solutions *state* for variable x then we can declare events as follows

$$\textsc{ree}\,(x,p) \overset{def}{=} state\,(x,n-1) - p < 0 \;\wedge\; state\,(x,n) - p \geq 0 \tag{3}$$

$$\textsc{fee}\,(x,p) \overset{def}{=} state\,(x,n-1) - p > 0 \;\wedge\; state\,(x,n) - p \leq 0 \tag{4}$$

$$\textsc{te}\,(t) \overset{def}{=} time\,(n-1) < t \;\wedge\; time\,(n) = t \tag{5}$$

whereby n is the index used in both sequences. The event REE is the so-called rising edge zero crossing and FEE is the falling edge zero crossing. The zero crossing functions of the solver ensure that $time(n)$ is an accurate approximation within user-defined bounds. The time event TE is generated as soon as the solver has exactly reached time t, whereby the solver ensures that the solution x in $state(x,n)$ at $time(n) = t$ is an accurate approximation. For our case study, we define two edge triggered events: REE (*level*, 3.0) and FEE (*level*, 2.0), whereby *level* is a shared continuous time variable that represents the height of the water level in the tank. This variable is declared on line 2 of Fig. 3 and line 4 of Fig. 5. An event is declared as a normal equation in 20-SIM [13] as shown in Fig. 4. In this example, we increment a simple event counter `eue` and inform the CT solver that the DE model needs to be updated, by setting the variable `fireDES`.

We use VDM++ [14] in this paper to describe the controller. We extended this notation in earlier work [15] such that the behavior of distributed embedded

```
// check for the upper water level limit
if (eventup(level - 3.0)) then
    eue = eue + 1;
    fireDES = true;
end;
```

Fig. 4. The REE (*level*, 3.0) event in 20-SIM

real-time systems can be analyzed by means of discrete event simulation. Here we assume a single processor system `cpu1` that executes the controller presented in Fig. 5. The shared continuous sensor and actuator variables *level* and *valve* are declared on Line 4 and 5. Whenever *level* is read, it contains the actual value of the corresponding continuous time variable on line 11 of Fig. 3. Similarly, whenever *valve* is assigned a value, it changes the state of `X0` in Fig. 3.

We demonstrate that two styles of control can be specified: event driven control and time triggered control. For event driven control, two asynchronous operations, *open* and *close* are defined in lines 8 and 11 respectively. The former will be the handler for the REE (*level*, 3.0) event and the latter is the handler for the FEE (*level*, 2.0) event. In other words, these asynchronous operations will be called automatically whenever the corresponding event fires. This will cause the creation of a new thread. This thread will die as soon as the operation is completed. In VDM++, all statements have a default duration, which can be redefined using the `duration` and `cycles` statements. The duration statement on line 9 states that opening the valve in this case takes 50 *msec*. The cycles statement on line 12 denotes that closing the valve takes 1000 cycles. Assuming this

```
01  class Controller
02
03  instance variables
04    static public level : real;
05    static public valve : bool := false  -- default is closed
06
07  operations
08    static public async open: () ==> ()
09    open () == duration(0.05) valve := true;
10
11    static public async close: () ==> ()
12    close () == cycles(1000) valve := false;
13
14    loop: () ==> ()
15    loop () ==
16      if level >= 3 then valve := true -- check high water mark
17      else if level <= 2 then valve := false; -- check low water mark
18
19  threads
20    periodic(1.0,0,0,1.0)(loop)
21
22  sync
23    mutex(open, close, loop)
24
25  end Controller
```

Fig. 5. The controller description in VDM++

class is deployed on a processor with a capacity of 100000 cycles per second, then executing `valve := false` will take 10 *msec*. Note that the result of the assignment is available *after* this time has passed. Time triggered control is provided by the *loop* operation in line 14-17. The `periodic` clause in line 20 states that the operation *loop* is called periodically, once per second, starting at $t = 1\,sec$. Note that we use the default statement durations here. Finally, the `mutex` clause on line 23 states that the three operations are declared mutually exclusive. This implies that only one operation call can be active at any time and they cannot be interrupted by each other. All threads that do not meet this requirement are blocked until the currently executing operation call is completed.

4 Tool Support

We implemented a discrete event simulator to execute VDM++ models as described in the previous section, as a proof of concept. We coupled this tool to the 20-SIM [13] continuous time simulator for dynamic systems. This tool has the ability to make calls to user-defined libraries from within the simulation. We implemented a simple DLL in C++ to exchange arbitrary sequences of double precision reals over a TCP/IP connection. The same library is used in the VDM++ simulator to set-up a connection. The progress of time in the simulators on either end of the connection is synchronized by exchanging the current time, time steps, actuator and sensor values and events, whereby the current time is always strict monotone increasing. In this section we will focus on the construction and use of the interface. In the next section we will look at the semantics in more detail.

The behavior of the interface is shown in the UML sequence diagram in Fig. 7. We use an XML configuration file to describe the information that is exchanged over the link, the interface is completely model independent. For brevity, we use an informal description as presented in Fig. 6. The keywords `sensor` and `actuator` are defined as perceived from the perspective of the discrete event simulator. Basically, we define a `sensor[]` array, an `actuator[]` array and an `event[]` array. These arrays provide the bindings for all variables and events. The `abort` keyword is used to stop the simulation, in addition to other tool specific stop criteria that may be defined, and gives control back to the user, for example to inspect the state of the model.

The XML configuration file is read by both simulations when the interface is started, indicated by `initialize` in Fig. 7. When a message is sent from

```
sensor[1] = cpu1.Controller‘level
actuator[1] = cpu1.Controller‘valve
event[1] = REE(level,3.0) -> cpu1.Controller‘open
event[2] = FEE(level,2.0) -> cpu1.Controller‘close
event[3] = TE(15.0) -> abort
```

Fig. 6. The interface configuration file

VDM++ to 20-SIM, indicated as `updateCT` in Fig. 7, the message contains the current time T, the target time step t_s, and the value of each defined actuator variable at T from `actuator[]`. So, for our case study only three values are exchanged in this direction for every step. Upon arrival, the operation `updateCTmodel` calls the continuous time solver and tries to perform the time step t_s. Either this time was reached or the solver stopped due to an event that occurred at t_r. When a message is sent from 20-SIM to VDM++, indicated as `updateDE` in Fig. 7, the message contains the current time T, the realized time step $t_r \leq t_s$, the value of each defined sensor variable at $T+t_r$ from `sensor[]`, followed by a monotone increasing counter for each declared `event[]`. This counter is incremented when the event occurred at $T+t_r$. This allows us to monitor the integrity of the interface. Several events can be detected at the same time, but an event can only occur once per iteration. Six values are offered when a message is sent from 20-SIM to VDM++ in this model. Upon arrival, the operation `updateDEmodel` processes all events, updates the shared continuous variables and performs a simulation step on the discrete event model, after which we iterate.

Figure 8 shows a simulation run for our case study, whereby we have disabled all state events. In other words, we are studying the periodic control loop behavior (lines 14-17 in Fig. 5). The top screen shows the evolution of the *level* sensor variable. The middle screen shows the evolution of the *valve* actuator variable. The bottom screen shows when the controller has been active, by means of a counter which is increased whenever the VDM++ model makes a time step. It resembles a staircase profile because the execution times of a single instruction are small compared to the changes in the water level. However, if we zoom in, we can actually see how much time is spent in the control loop. Notice that the

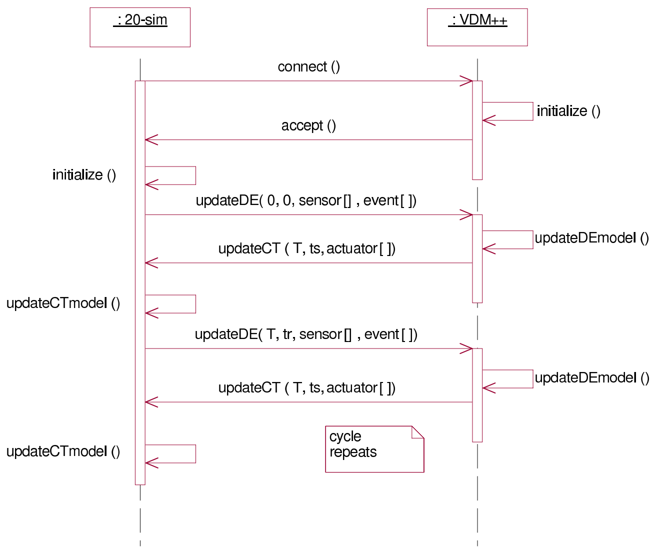


Fig. 7. Tool interface behavior as a UML sequence diagram

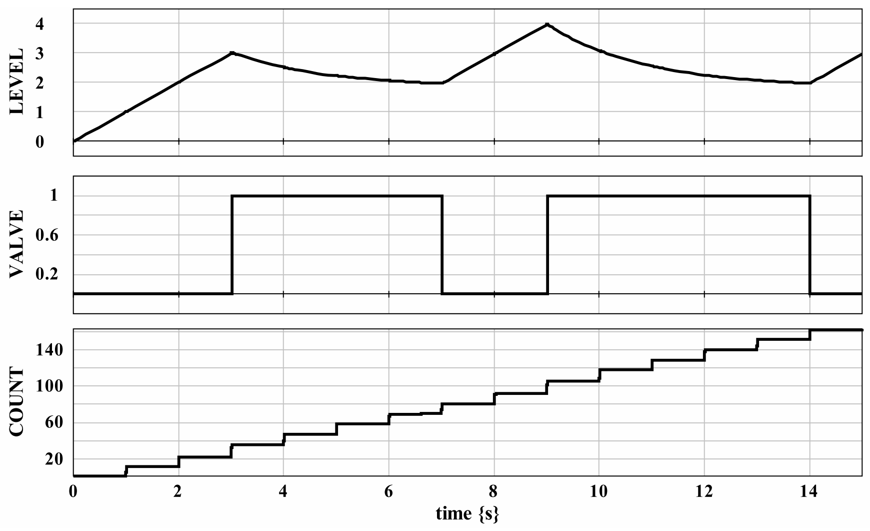


Fig. 8. Visualization of a co-simulation run of the water tank case study

discrete controller is indeed invoked every second, but the control actions, for example at $t = 4\,sec$ are slightly delayed, as expected. Moreover, observe that the valve was not opened at $t = 8\,sec$ because *level* was 2.96 at that time. The overshoot would have been substantially smaller if event based control was used here. We can change many system parameters in the discrete event simulator and observe their impact, such as the processor speed, task switch overheads, and the scheduling policy, without modifying the model shown in Fig. 5. Similarly, we can change parameters in 20-SIM, such as the input flow rate, the liquid density, the resistance of the valve exit, etc.

5 Reconciled Operational Semantics

There are many techniques available from computer science that can be used to create discrete event models. Two-phase labeled transition systems are commonly used, whereby state and time transitions are explicitly distinguished. Assuming some initial state, in the first phase, the successor state is computed and then time elapses in the second phase after which the process is repeated. We have presented an abstract operational semantics for distributed embedded real-time systems in VDM++ in [15] which is also based on this approach. In this paper, we extend this abstract formal semantics to allow for consistent co-simulation with continuous time models. The tool support described in the previous section conforms to the formal operational semantics presented here. One of the key features of our work is that state modifications computed in phase one are made visible after the time step in phase two has been completed, in order to guarantee consistency in the presence of shared continuous variables and arbitrary interleaving of multiple, concurrent, labeled transition systems.

The main aim of the operational semantics is to formalize the interaction between the discrete event simulator, which executes a control program, and a solver for a continuous time plant model. Hence we have omitted many details of the VDM++ model such as the links between nodes, message transfer along these links, the definition of classes, including explicit definitions of synchronous and asynchronous operations, guards and a concept to define periodic threads. The operational semantics of these concepts can be found in [15]. In contrast with this previous work, we will focus in this section on communication by means of global variables and events, since this is used to model the interaction between continuous time and discrete event models. In Sect. 5.1 we define the syntax of a simple imperative language which serves as an illustration of the basic concepts, without trying to be complete. The operational semantics of this language is defined in Sect. 5.2.

5.1 Syntax

The distributed architecture of an embedded control program can be represented by so-called nodes. Let *Node* be the set of node identities. Nodes are used to represent computation resources such as processors. On each node a number of concurrent threads are executed in an interleaved way. In addition, execution is interleaved with steps of a solver.

Threads can be created dynamically, e.g., to deal with events received from the solver. Let *Thread* be the set of thread identities, including dormant threads that can be made alive when a new thread is created. Function $node : Thread \rightarrow Node$ denotes on which node each thread is executing. Each thread executes a sequential program, that is, a statement expressed in the language of Table 1.

Let *Value* be a domain of values, such as the real numbers $\mathbb{R}$. Assume given a set of variables $Var = InVar \cup OutVar \cup LVar$ where $InVar$ is the set of input/sensor variables, $OutVar$ is the set of output/actuator variables, and $LVar$ a set of local variables. The input and output variables are global and shared between all threads and the continuous model. Hence, they can also be accessed by the solver, which may read the actuator variables and write the sensor variables. Let $IOVar = InVar \cup OutVar$. Let $Time = \mathbb{R}$ be the time domain. The syntax of our sequential programming language is given in Table 1, with $c \in Value$, $x \in Var$, and $d \in Time$.

The execution of basic statements such as skip and assignment $x := e$ takes zero time, except for the **duration**(d) statement which represents a time step of d time units. For each thread, any sequence of statements between two successive duration statements is executed atomically in zero time. However, the execution of such a sequence might be interleaved with statements of other threads or a step of the solver. Concerning the shared IO-variables in $IOVar$ this means that we have to ensure atomicity explicitly. Hence, we introduce a kind of *transaction* mechanism to guarantee consistency in the presence of arbitrary interleaving of steps. Thread *thr* is only allowed to modify IO-variable x if there is no transaction

Table 1. Syntax of Statements

Value Expression	$e ::= c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Boolean Expression	$b ::= e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Statement	$S ::= \textbf{skip} \mid x := e \mid \textbf{duration}(d) \mid S_1 \,;\, S_2 \mid$ $\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi} \mid \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}$

in progress by any other thread. The transaction is committed as soon as the thread performs a time step. This will be explained in detail in Defs. 2 and 5.

Let *SeqProg* be the set of sequential programs of the form $S\ ;\ E$, where E is an auxiliary statement which is used to denote termination of a thread.

The solver may send events to the control program. Let *Event* be a set of events. They may be defined by the primitives $\textsc{ree}\,(x, p)$, $\textsc{fee}\,(x, p)$, and $\textsc{te}\,(t)$, as proposed in Eqs. 3-5. Assume that an event handler has been defined for each event, i.e., a sequential program, and a node on which this statement has to be executed (as a new thread), denoted by the function $\mathit{evhdlr} : \mathit{Event} \rightarrow \mathit{SeqProg} \times \mathit{Node}$.

5.2 Operational Semantics

To define the operational semantics, we first introduce a *configuration* C in Def. 1 to capture the state of affairs at a certain point in the execution of our model. Next, we define the so-called *variant* of a configuration in Def. 2. The notion of a *step*, denoted by $C \longrightarrow C'$ for configurations C and C', is defined in Def. 3, using Defs. 4, 5, and 6. This finally leads to a set of runs of the form $C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \cdots$ in Def. 7, which provides the abstract formal operational semantics of simulating a control program in parallel with a solver of a continuous time model.

Definition 1 (Configuration). A *configuration* C contains the following fields:

- $\mathit{instr} : \mathit{Thread} \rightarrow \mathit{SeqProg}$
  the remaining program to be executed by each thread.
- $\mathit{curthr} : \mathit{Node} \rightarrow \mathit{Thread}$
  yields for each node the currently executing thread.
- $\mathit{status} : \mathit{Thread} \rightarrow \{\mathit{dormant}, \mathit{alive}\}$
  thread status; a thread can be created by making a dormant thread alive.
- $\mathit{lval} : \mathit{LVar} \times \mathit{Thread} \rightarrow \mathit{Value}$
  denotes the value of each local variable for each thread.
- $\mathit{ioval} : \mathit{IOVar} \rightarrow \mathit{Value}$
  denotes the committed value of each sensor and actuator variable.
- $\mathit{modif} : \mathit{IOVar} \times \mathit{Thread} \rightarrow \mathit{Value} \cup \{\perp\}$
  to denote the values of sensor and actuator variables that have been modified by a thread and for which the transaction has not yet been committed (by executing a duration statement). The symbol $\perp$ denotes that the value is

undefined, i.e., the thread did not modify the variable in a non-committed transaction.

– $now : Time$ to denote the current time. □

For a configuration C, we use the notation $C(f)$ to obtain its field f, such as $C(instr)$. We define a few suitable abbreviations:

– $cur(C, n)$ denotes the current thread on node n, i.e. $C(curthr)(n)$
– $exec(C, thr)$ expresses that thr is executing, i.e., there exists an $n \in Node$ such that $cur(C, n) = thr$.

We define the notion of a variant to express configuration modifications.

Definition 2 (Variant). The *variant* of a configuration C with respect to a field f and a value v, denoted by $C\,[f \mapsto v]$, is defined as

$$(C\,[f \mapsto v])(f') = \begin{cases} v & \text{if } f' = f \\ C\,(f') & \text{if } f' \neq f \end{cases} \tag{6}$$

Similarly for field parts, such as variants of mapping *ioval*. □

We define the value of an expression e in a configuration C which is evaluated in the current thread on a node n, denoted by $[\![\, e \,]\!](C, n)$. The main point is the evaluation of a variable:

$$[\![\, x \,]\!](C, n) = \begin{cases} C(modif)(x, cur(C, n)) & \text{if } x \in IOVar, C(modif)(x, cur(C, n)) \neq \bot \\ C(ioval)(x) & \text{if } x \in IOVar, C(modif)(x, cur(C, n)) = \bot \\ C(lval)(x, cur(C, n)) & \text{if } x \in LVar \end{cases}$$

The other cases are trivial, e.g., $[\![\, e_1 \times e_2 \,]\!](C, n) = [\![\, e_1 \,]\!](C, n) \times [\![\, e_2 \,]\!](C, n)$ and $[\![\, c \,]\!](C, n) = c$. It is also straightforward to define when a Boolean expression b holds in the current thread of a configuration C on a node n, denoted by $[\![\, b \,]\!](C, n)$. For instance, $[\![\, e_1 < e_2 \,]\!](C, n)$ iff $[\![\, e_1 \,]\!](C, n) < [\![\, e_2 \,]\!](C, n)$, and $[\![\, \neg b \,]\!](C, n)$ iff not $[\![\, b \,]\!](C, n)$.

Definition 3 (Step). $C \longrightarrow C'$ is called a *step* if and only if it corresponds to the execution of a statement (Def. 4), performing a time step (Def. 5), or a context switch (Def. 6). □

Definition 4 (Execute Statement). A step $C \longrightarrow C'$ corresponds to the execution of a statement if and only if there exists at least one executing thread thr with $exec(C, thr)$ such that $C(instr)(thr) = S_1\ ;\ S_2$, allowing $S_2 = E$, and one of the following clauses holds:

– $S_1 =$ **skip**. The skip statement does not have any effect except that the statement is removed from the instruction sequence
$C' = C[instr(thr) \mapsto S_2]$.
– $S_1 = x := e$. We distinguish two cases, depending on the type of variable x.

- If $x \in \mathit{IOVar}$ we require that there is no transaction in progress by any other thread: for all thr' with $\mathit{thr}' \neq \mathit{thr}$ we have $C(\mathit{modif})(x, \mathit{thr}') = \bot$. Then the value of e is recorded in the modified field of thr:
  $C' = C[\mathit{instr}(\mathit{thr}) \mapsto S_2, \mathit{modif}(x, \mathit{thr}) \mapsto [\![e]\!](C, n)]$

  As we will see later, all values belonging to thread thr in C(modif) are removed and bound to the variables in $C(\mathit{ioval})$ as soon as thread thr completes a time step (Def. 5). This corresponds to the intuition that the result of a computation is available only at the end of the time step that reflects the execution of a piece of code.
- If $x \in \mathit{LVar}$ then we change the value of x in the current thread:
  $C' = C[\mathit{instr}(\mathit{thr}) \mapsto S_2, \mathit{lval}(x, \mathit{thr}) \mapsto [\![e]\!](C, n)]$

– $S_1 = \textbf{if}\ b\ \textbf{then}\ S_{11}\ \textbf{else}\ S_{12}\ \textbf{fi}$. If $[\![b]\!](C, n)$ then we have
  $C' = C[\mathit{instr}(\mathit{thr}) \mapsto S_{11}\ ;\ S_2]$, otherwise $C' = C[\mathit{instr}(\mathit{thr}) \mapsto S_{12}\ ;\ S_2]$.

– $S_1 = \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}$. If $[\![b]\!](C, n)$ then we have
  $C' = C[\mathit{instr}(\mathit{thr}) \mapsto S\ ;\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ ;\ S_2]$, otherwise we obtain
  $C' = C[\mathit{instr}(\mathit{thr}) \mapsto S_2]$.

Observe that the execution of these statements does not affect now, that is, $C(\mathit{now}) = C'(\mathit{now})$. $\square$

Definition 5 (Time Step). A step $C \longrightarrow C'$ is called a *time step* only if all current threads are ready to execute a duration instruction or have terminated. More formally, for all thr with $\mathit{exec}(C, \mathit{thr})$, $C(\mathit{instr})(\mathit{thr})$ is of the form $\textbf{duration}(d)\,;\,S$ or equals E. Then the definition of a time step consists of three parts: (1) the definition of the requested duration of the time step, (2) the execution of this time step by the solver, leading to intermediate configuration C_s (3) updating all durations of all current threads, dealing with events generated by the solver, and committing all variables of the current threads.

1. Time may progress with a number of time units which is smaller than or equal to all durations of all current threads. Hence, the requested length of the time step is defined by
   $t_s = \mathit{min}\{d \mid \exists\, \mathit{thr}, S : \mathit{exec}(C, \mathit{thr}) \wedge C(\mathit{instr})(\mathit{thr}) = \textbf{duration}(d)\ ;\ S\}$.
2. If $t_s > 0$ the solver tries to execute a time step of length t_s in configuration C. Concerning the variables, the solver will only use the *ioval* field, ignoring the *lval* and *modif* fields. It will only read the actuator variables in OutVar and it may write the sensor variables in InVar in field *ioval*. As soon as the solver generates one or more events, its execution is stopped. This leads to a new configuration C_s and a set of generated events *EventSet*. Since the solver takes a positive time step, $C(\mathit{now}) < C_s(\mathit{now}) \leq C(\mathit{now}) + t_s$, and if $C_s(\mathit{now}) < C(\mathit{now}) + t_s$ then $\mathit{EventSet} \neq \emptyset$. Moreover, $C_s(f) = C(f)$ for $f \in \{\mathit{instr}, \mathit{curthr}, \mathit{status}, \mathit{lval}, \mathit{modif}\}$.

   If $t_s = 0$ then the solver is not executed and $C_s = C$, $\mathit{EventSet} = \emptyset$. This case is possible because we allow $\textbf{duration}(0)$ to commit variable changes, as shown in the next point.

3. Starting from configuration C_s and *EventSet*, next the durations are decreased with the actual time step performed, new threads are created for the event handlers, and finally for threads with zero durations the transactions are committed.

   Let $t_r = C_s(now) - C(now)$ be the time step realized by the solver. For each event $e \in EventSet$ with $evhdlr(e) = (S_e, n_e)$, let thr_e be a fresh - not yet used - thread identity with status *dormant* and $node(thr_e) = n_e$.

   We define an auxiliary function $NewInstr(C, t_r) : Thread \rightarrow SeqProg$ which decreases durations, removes zero durations, and installs event handlers:
   $NewInstr(C, t_r)(thr) =$
   $$\begin{cases} duration(d - t_r)\ ;\ S & \text{if } exec(C, thr),\ C(instr)(thr) = duration(d)\ ;\ S, \\ & \quad \text{and } d > t_r \\ S & \text{if } exec(C, thr) \text{ and } C(instr)(thr) = duration(t_r)\ ;\ S \\ S_e & \text{if } thr = thr_e \text{ for some } e \in EventSet \\ C(instr)(thr) & \text{otherwise} \end{cases}$$
   Next define a function to awake the new threads for event handlers:
   $$NewStatus(C)(thr) = \begin{cases} alive & \text{if } thr = thr_e \text{ for some } e \in EventSet \\ C(status)(thr) & \text{otherwise} \end{cases}$$
   Let $ActDurZero = \{thr \mid exec(C, thr) \text{ and } C(instr)(thr) = duration(t_r)\ ;\ S\}$ be the set of threads which will have a zero duration after this time step. For these threads the transactions are committed and the values of the modified variables are finalized. This is defined by two auxiliary functions:
   $NewIoval(C)(x) =$
   $$\begin{cases} v & \text{if } \exists\, thr \in ActDurZero \text{ and } C(modif)(x, thr) = v \neq \bot \\ C(ioval)(x) & \text{otherwise} \end{cases}$$
   Note that at any point in time at most one thread may modify the same global variable in a transaction. Hence, there exists at most one thread satisfying the first condition of the definition above, for a given variable x.
   The next function resets the modified field.
   $$NewModif(C)(x, thr) = \begin{cases} \bot & \text{if } thr \in ActDurZero \\ C(modif)(x, thr) & \text{otherwise} \end{cases}$$
   Then $C' = C_s[\ instr \mapsto NewInstr(C_s, t_r),\ status \mapsto NewStatus(C_s),$
   $ioval \mapsto NewIoval(C_s),\ modif \mapsto NewModif(C_s)]$
   Observe that $C'(now) = C_s(now) = C(now) + t_r$ with $t_r \leq t_s$. □

Definition 6 (Context Switch). A step $C \longrightarrow C'$ corresponds to a context switch iff there exists a thread *thr* which is alive and not running, and which has a non-empty program, that is, $\neg exec(C, thr)$, $C(status)(thr) = alive$, and $C(instr)(thr) = S \neq E$. Then *thr* becomes the current thread and a duration of δ_{cs} time units is added to represent the context switching time:
$C' = C[\ instr(thr) \mapsto \mathbf{duration}(\delta_{cs})\ ;\ S,\ curthr(node(thr)) \mapsto thr\ ]$ □

Note that more than one thread may be eligible as the current thread on a node at a certain point in time. In that case, a thread is chosen nondeterministically in our operational semantics. Fairness constraints or a scheduling strategy may be added to enforce a particular type of node behavior, such as for example rate monotonic scheduling.

Definition 7 (Operational Semantics). The operational semantics of our model is the set of execution sequences of the form $C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \cdots$, where each pair $C_i \longrightarrow C_{i+1}$ is a step (Def. 3) and the initial configuration C_0 all current threads are *alive* and the *modif* field is $\perp$ everywhere. Finally, to avoid Zeno behaviour, we require that for any point of time t there exists a configuration C_i in the sequence with $C_i(\mathit{now}) > t$. $\square$

6 Concluding Remarks

A multidisciplinary modeling approach shall provide sufficient means of abstraction to support all mono-disciplinary views in order to be industrially applicable. A solid semantic foundation of the combination of these views is required to support meaningful and reliable analysis of the heterogenous model. We believe that this can be achieved by taking a "best of both worlds" approach whereby the software discipline uses a formal specification technique. Firstly because it provides abstraction mechanisms that allow high-level specification and secondly because its well-defined semantics provides a platform independent description of the model behavior that can be analyzed properly. Software models as advocated by IBM Rational Technical Developer and I-Logix Rhapsody are, in our opinion, not suited for this purpose in particular because they lack a suitable notion of abstraction, time and deployment. We showed how tool integration can be achieved based on the formal semantics proposed in this paper, which we applied to a case study. Note however that the approach taken here is not specific to any tool in particular. Our approach has been applied to a larger case study: the distributed controller of a paper path of a printer [16].

Nicolescu et al [17] propose a software architecture for the design of continuous time / discrete event co-simulation tools for which they provide an operational semantics in [18]. Our work is in fact an instantiation of that architecture, however, with a difference. Their approach is aimed at connecting multiple simulators on a so-called simulation bus, whereas we connect two simulators using a point-to-point connection. They use Simulink and SystemC whereas we use 20-SIM and VDM++ to demonstrate the concept. The type of information exchanged over the interfaces is identical (the state of continuous variables and events). They have used formal techniques to model properties of the interface, whereas we have integrated the continuous time interface into the operational semantics of a discrete event system. We believe that our approach is stronger because a weak semantics for the discrete event model may still yield unexpected simulation results even though the interface is proven to work consistently. An in-depth comparison of both approaches is subject for further study.

The interface between the continuous time and discrete event models seems to be convenient when resilience of a system is studied. Early experiments performed in collaboration with Zoe Andrews at the Centre for Software Reliability at Newcastle University have shown that it is possible to use this interface for fault injection. Values and events exchanged over this interface can be dropped, inserted, modified, delayed and so on to represent the failure mode of a sensor or actuator, such as for example "stuck at x". The advantage of this approach is that the failure model can remain orthogonal to the continuous time and the discrete event models. These system models need no longer be obscured by explicit failure mode modeling in either plant or controller, which usually clobbers the specification. We certainly plan to explore this further.

In summary, the approach is to bring realistic time-aware models of software, executed on a possibly distributed hardware architecture, into the realm of control engineering without enforcing a certain model of computation a priori. We propose to use formal specification techniques to provide suitable software models required for this approach, mainly in order to manage complexity such that small, abstract and high-level models can be created. This is essential in the early phases of the system design life-cycle, where changes are likely to occur while working under severe time pressure. We provide a system level approach for modeling computation, communication and control with support and flexibility for the decision making during the early phases of the system-design life cycle, whereby the trade-offs can be investigated by co-simulation.

Acknowledgments. The authors wish to thank Job van Amerongen, Zoe Andrews, Peter van den Bosch, Erik Gaal, Peter Gorm Larsen, Frits Vaandrager and the anonymous reviewers for their valuable comments to this paper and support for this work.

References

1. Boderc: Model-based design of high-tech systems. Final report. Embedded Systems Institute, P.O. Box 513, 5600 MB Eindhoven, NL (2006) Available on-line at `http://www.esi.nl/boderc`
2. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006)
3. Henriksson, D.: Flexible Scheduling Methods and Tools for Real-Time Control Systems. PhD thesis, Lund Institute of Technology, Department of Automatic Control (2003) `http://www.control.lth.se/truetime/`
4. Douglas, B.P.: Real-Time UML Workshop for Embedded Systems. Embedded Technology. Newnes. Elsevier, Amsterdam (2007)
5. Hooman, J., Mulyar, N., Posta, L.: Coupling Simulink and UML models. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 304–311. Springer, Heidelberg (2004)

6. Davis, J., Galicia, R., Goel, M., Hylands, C., Lee, E., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J., Smyth, N., Tsay, J., Xiong, Y.: Ptolemy-II: Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL No. M99/40, University of California at Berkeley (1999)
7. Karnopp, D.C., Margolis, D.L., Rosenberg, R.C.: System Dynamics: Modeling and Simulation of Mechatronic Systems, 3rd edn. Wiley-Interscience, Chichester (2000)
8. Breedveld, P.: Multibond-graph elements in physical systems theory. Journal of the Franklin Institute 319, 1–36 (1985)
9. Ledin, J.: Simulation Engineering - Build Better Embedded Systems Faster. Embedded Systems Programming. CMP Books (2001)
10. Hairer, E., Nørsett, S.P., Gerhard., W.: Solving ordinary differential equations I: Nonstiff problems, 2nd edn. Springer, Heidelberg (1993)
11. Hairer, E., Wanner, G.: Solving ordinary differential equations II: Stiff and differential-algebraic problems, 2nd edn. Springer, Heidelberg (1996)
12. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Kluwer Academic Publishers, Dordrecht (1999)
13. ControlLab Products: 20-sim (2006) http://www.20sim.com
14. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer, Heidelberg (2005) http://www.vdmbook.com
15. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and validating distributed embedded real-time systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 147–162. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11813040_11
16. Visser, P., Verhoef, M., Broenink, J., Hooman, J.: Co-simulation of continuous-time/discrete-event systems as vehicle for embedded system design trade-off's (Submitted, 2007)
17. Nicolescu, G., Boucheneb, H., Gheorghe, L., Bouchhima, F.: Methodology for efficient design of continuous/discrete-events co-simulation tools. In: Anderson, J., Huntsinger, R. (eds.) High Level Simulation Languages and Applications - HLSLA. SCS, San Diego, CA, pp. 172–179 (2007)
18. Gheorghe, L., Bouchhima, F., Nicolescu, G., Boucheneb, H.: Formal definitions of simulation interfaces in a continuous/discrete co-simulation tool. In: Proc. IEEE Workshop on Rapid System Prototyping, pp. 186–192. IEEE Computer Society Press, Los Alamitos (2006) http://doi.ieeecomputersociety.org/10.1109/RSP.2006.18