

Formalizing Causal Block Diagrams for Modeling a Class of Hybrid Dynamic Systems

Ben Denckla
Denckla Consulting
1607 S. Holt Ave.
Los Angeles, CA 90035, USA
bdenckla@alum.mit.edu

Pieter J. Mosterman
The MathWorks, Inc.
3 Apple Hill Dr.
Natick, MA 01760, USA
pieter.mosterman@mathworks.com

Abstract—This paper attempts to formalize the semantics of causal block diagrams, a language that is extensively used in the design of technical systems. The formalization is based on *lambda calculus*, and implemented in the declarative functional language Haskell. Specifically, the combination of discrete-time and continuous-time computations, *hybrid dynamic systems*, is concentrated on. It shows how in many cases this combination causes multi-rate computations and so transition semantics between the two types of computations are strictly necessary. A loose interpretation is shown to result in an implementation that is amenable to error.

I. INTRO

With the embedding of computing power into control systems to implement computationally intensive functionality, the fields of control systems engineering and computer-based systems have become intertwined. This has led to software engineering research in the field of control system integration [4] and control system design research in the field of software engineering [5]. It is desirable to bring together these different strands of research and integrate software engineering notions into the control systems community, as well as developing a control engineering understanding in the computer-based systems community [8], [9].

The field of *hybrid dynamic systems*¹ integrates the control systems and computer science disciplines [15]. Hybrid dynamic systems combine the discrete dynamics typically implemented by embedded computation with the continuous dynamics that often best describe the physics of the process under control.

To be more precise, three classes of execution can be distinguished [1]: (i) discrete-event behavior, (ii) discrete-time behavior, and (iii) continuous-time behavior. This paper concentrates on integrating the latter two in the context of *causal block diagrams* [13]. As the basic language of Simulink[®] [14], such diagrams are prevalent in industry for embedded control design purposes and have shown to be convenient in modeling discrete-time as well as continuous-time systems.

The challenge in embedded control system design is to have precisely defined languages so the controller design can be analyzed accurately. This is, however, partially at

¹The adjective *dynamic* is included to distinguish from other hybrid systems such as those that combine neural networks and fuzzy logic and systems that combine mechanical and electric drivetrains.

odds with demands from industry. On the one hand, a very precise and specific language is needed, on the other hand, the language should not become overly constrictive and unwieldy to satisfy the preciseness constraint.

Because powerful, the definition of the causal block diagram language as employed by Simulink allows implicit conversions that need to be made explicit in a strict semantic context. This paper aims to establish such a formal semantics of causal block diagrams. As such, it supports automatic code generation, which will bring together the control system design engineers and the computer-based systems experts. It allows their different backgrounds to merge and communication in the common high-level language of causal block diagrams, extended with architectural features that are required to properly engineer the eventual software.

Section II first provides the formal definitions of the block diagram language. The abstract syntax of the elements of a block diagram are defined using a class diagram, while the semantics are defined using functions in their most general form. In Section III the definitions are made concrete by an example, where lambda calculus [10] is used to define the functions. An implementation of the block diagram example using Haskell [6] is given and a simulator in Haskell is provided as well. In Section IV the block diagram executions are then endowed with the notion of time, where a temporal semantics is associated with the previously developed atemporal simulator. In Section V the notion of continuous-time is added to the block diagram language. Section VI analyzes the interaction between continuous-time and discrete-time behaviors and Section VII presents the conclusions of this work.

II. DEFINITIONS

The causal block diagram language is defined by its syntax and semantics.

A. Abstract Syntax

A model of the abstract syntax of a causal block diagram language is shown in Fig. 1. The concrete syntax is not formally given as it is very much tool-specific and not of interest to this treatise. The concrete syntax that is used in the proceedings is that of Simulink.

A block diagram consists of entities and directed relations. The entities can be blocks, input ports and output

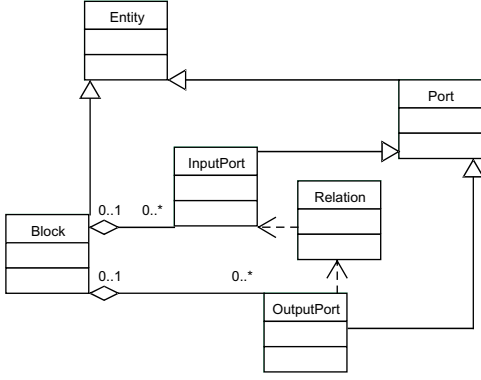


Fig. 1. Model of the abstract syntax of a block diagram.

ports. Relations are directed and connected to input ports and output ports. An input port can only be the destination of a relation, whereas an output port can only be the source of a relation.

Input ports and output ports can be bound to exactly one block or no block at all, indicated by the $0..1$ cardinality. In turn, each block can have zero or an arbitrary number of input and output ports bound to it ($0..*$ cardinality).

The block is the basis for each block diagram. It can be connected to other blocks by its input and output ports. Input and output ports that are not associated with a block represent input and output that is external to the block diagram.

B. Static Semantics

To define the semantics of the block diagram language, a mathematical representation is given for each of the concrete entities in the abstract syntax: `Block`, `InputPort`, `OutputPort`, and `Relation`. Note that, though the function domains will be defined to be the complex values, the domains can be arbitrarily chosen. Instead of numerical values, symbols could be chosen as well. This notion is not further discussed in this paper.

1) *Block*: A `Block` can have `InputPorts` and `OutputPorts` associated with it. Each `InputPort` is bound to a variable u_i , $i \in [1..l]$ that is considered input to the `Block`, and each `OutputPort` is bound to a variable y_j , $j \in [1..m]$ that is considered output of the `Block`. In addition, a `Block` binds variables x_k and z_k , $k \in [1..n]$ to non-visible input and output that connects to the execution engine.

The behavior of a `Block` is defined by a function, f , that maps the space of explicit input, U , and the space of implicit input, X , onto the space of explicit output, Y , and the space of implicit output, Z .

Definition 1 (Block): A `Block` is a tuple $\langle U, X, Y, Z, f \rangle$ with $f : U \times X \rightarrow Y \times Z$, and $U \in (\mathbb{R} \times \mathbb{S})^l$, $X, Z \in (\mathbb{R} \times \mathbb{S})^n$, and $Y \in (\mathbb{R} \times \mathbb{S})^m$, where $l, m, n \in \mathbb{N}$.

2) *InputPort*: An `InputPort` that is not associated with a `Block` binds an external output variable, v_j , to an explicit output variable, y_j , $j \in [1..m]$.

Definition 2 (InputPort): An unbound `InputPort` is a pair $\langle \Upsilon, Y \rangle$ with $(\forall j \in [1..m])v_j = y_j$, and $\Upsilon, Y \in (\mathbb{R} \times \mathbb{S})^m$, and $m \in \mathbb{N}$.

3) *OutputPort*: An `OutputPort` that is not associated with a `Block` binds an external input variable, ψ_j , to an explicit input variable, u_i , $i \in [1..l]$.

Definition 3 (OutputPort): An unbound `OutputPort` is a pair $\langle U, \Psi \rangle$ with $(\forall i \in [1..l])u_i = \psi_i$, and $U, \Psi \in (\mathbb{R} \times \mathbb{S})^l$, and $l \in \mathbb{N}$.

4) *Relation*: A `Relation` binds an explicit output variable, y_i , to an explicit input variable, u_i , with $i \in [1..p]$ and $p \leq m \wedge p \leq n$.

Definition 4 (Relation): A `Relation` is a pair $\langle U, Y \rangle$ with $(\forall i \in [1..p])u_i = y_i$, and $U, Y \in (\mathbb{R} \times \mathbb{S})^p$, and $p \in \mathbb{N}$.

C. Block Diagram

The entire block diagram can then be defined as a function that operates on the external and implicit input to the block diagram and returns the external and implicit output of the block diagram.

Definition 5 (Block Diagram): A `BlockDiagram` is a tuple $\langle \Upsilon, X, \Psi, Z, \phi \rangle$ with $\phi : \Upsilon \times X \rightarrow \Psi \times Z$ and $\Upsilon \in (\mathbb{R} \times \mathbb{S})^m$, $X, Z \in (\mathbb{R} \times \mathbb{S})^n$, $\Psi \in (\mathbb{R} \times \mathbb{S})^l$, and $l, m, n \in \mathbb{N}$.

Note that the definition of the static semantics of a block diagram coincides with the definition of a block (Definition 1) and so a block diagram can be considered a block in its own right. This allows for a hierarchical decomposition of blocks into block diagrams, but this notion is beyond the scope of this paper.

D. Execution Manager

An `ExecutionManager` repeatedly executes the block diagram function, ϕ , and assigns it an interpretation as an evolving behavior. The execution manager provides the external and implicit input to the block diagram and obtains the external and implicit output of the block diagram. In its most general form, this behavior is an iteration that modifies the output of ϕ at each iteration to produce a new input to ϕ . The new input to ϕ is computed by a function θ .

Definition 6 (Execution Manager): The `ExecutionManager` consist of a function $\theta : \Psi \times Z \rightarrow \Upsilon \times X$ that is repeatedly applied and includes a block diagram function $\phi : \Upsilon \times X \rightarrow \Psi \times Z$.

In discrete-time execution, the function θ typically assigns z_i to x_i , i.e., $(\forall i \in [1..l])(x_i = z_i)$, to compute the new input to ϕ :

$$\theta : \begin{cases} \langle \psi, z \rangle = \phi(v, x) \\ x = z \end{cases} \quad (1)$$

where $v \in \Upsilon$, $\psi \in \Psi$, $x \in X$, and $z \in Z$ and $m = l$.

The repeated execution of θ results in a *trajectory* or *behavior* of the block diagram output.

Definition 7 (Behavior): A *behavior* is a sequence of pairs $\langle \Upsilon, X \rangle_i$ with $i \in [1..\mu]$ and $\mu \in \mathbb{N}$.

III. A HASKELL IMPLEMENTATION

A block diagram simulator defines the block diagram as well as the execution manager using the functional language Haskell [6]. Note that in related work, Haskell is used to define noncausal models [11].

A. An Example

To make the definitions and their elements in Section II concrete, consider the block diagram in Fig. 2. This block diagram consists of a *Constant* block and an input port, *In*, that are connected to a *Sum* block. The *Sum* block connects to a *Delay* block. The *Delay* block connects to both a *Scope* and output port, *Out*, which illustrates the destination cardinality of relations being more than 1.

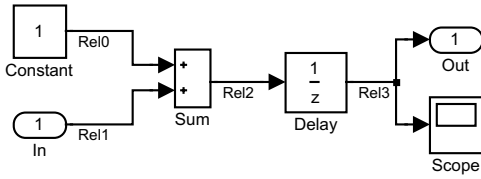


Fig. 2. A block diagram example.

In terms of the abstract syntax, this block diagram consists of four `Blocks` (*Constant*, *Sum*, *Delay*, *Scope*), four `OutputPorts` (one free one called *In*, and one associated with each of the *Constant*, *Sum*, and *Delay* blocks), five `InputPorts` (one free one called *Out*, two associated with the *Sum* block, and one associated with each of the *Delay* and *Scope* blocks), and five `Relations`. There are three relations with destination cardinality one, marked *Rel0* through *Rel2*, while there is one relation with destination cardinality two, marked *Rel3*.

To execute this block diagram, each of the blocks has to be defined in terms of a function that allows *lazy evaluation*. Lazy evaluation (as opposed to strict evaluation) allows a function evaluation without all the input arguments being available. This matches blocks in causal block diagrams perfectly, as it may be beneficial to evaluate the one function that captures the semantics of a block in multiple stages. For example, in Simulink one block is defined by multiple functions such as *mdlOutputs* and *mdlUpdate* that are evaluated strictly [3], [7].

B. The Block Diagram Function

Because of its declarative nature and lazy evaluation, in this work, lambda calculus [10] is used to capture the semantics of each of the blocks. Haskell is mostly a syntactic sophistication of the lambda calculus.

The definitions of the block constructors for the block diagram in Fig. 2 are

```
c_AddAdd = uncurry (+)
c_UnitDelay x0 = (\x u -> (u, x), x0)
c_Constant c = (\x -> (x, x), c)
c_Recorder = (\x u -> x ++ [u], [])
```

The Haskell expression $\backslash x \rightarrow y$ is an unnamed function with formal argument x and body y . It is analogous to

$\lambda x.y$ in the lambda calculus. For example, the function for the *Delay* block shows two arguments, the state x and the input u . The function body appears after the \rightarrow operator and for *Delay*, it shows how it returns the state as new output and the input as the new state (the ordering of the x and u pair is reversed). The *Constant* function operates similarly, except that it does not update its state. The *Recorder* function shows how it adds the input u to the list of stored values x . Addition is defined as the Haskell function ‘+’ in its ‘uncurried’ form, i.e., it takes one pair as an argument instead of two separate values.

The Haskell code that is generated for the block diagram in Fig. 2 is straightforward:

```
s = (f_toplevel, x0_toplevel)
  where

-- Subsystem construction
-- e.g. (f_System, x0_System) = c_SystemConstructor p x0
(f_Constant, x0_Constant) = c_Constant 1
(f_Delay, x0_Delay) = c_UnitDelay 1
(f_Scope, x0_Scope) = c_Recorder
f_Sum = c_AddAdd

-- Initial state computation and state advance function
x0_toplevel = (x0_Constant, x0_Delay, x0_Scope)
f_toplevel (x_Constant, x_Delay, x_Scope) y_In =
  ((xp_Constant, xp_Delay, xp_Scope), u_Out)

  where

-- Connections between ports
-- e.g. u_System1_inport = y_System2_outport
u_Delay_input = y_Sum_output
u_Scope_input = y_Delay_output
u_Sum_input1 = y_Constant_output
u_Sum_input2 = y_In
u_Out = y_Delay_output

-- Bindings of ports to functions
-- e.g. y_System_outport = f_System u_System_inport
(xp_Constant, y_Constant_output) =
  f_Constant x_Constant
(xp_Delay, y_Delay_output) =
  f_Delay x_Delay u_Delay_input
xp_Scope = f_Scope x_Scope u_Scope_input
y_Sum_output = f_Sum (u_Sum_input1, u_Sum_input2)
```

C. The Execution Manager

The execution manager is also implemented in Haskell:

```
execManager (f,x) [] = (x,[])
execManager (f,x) (u:us) = (xFinal, (x,y):restOfXYPairs)
  where
  (xp,y) = f x u
  (xFinal,restOfXYPairs) = execManager (f,xp) us
```

The first line (base case of the recursion) says: if any system is run on an empty input sequence $[]$, return the initial state as the final state and an empty sequence of external output and implicit output pairs. The second and following lines generate the trajectory. When applied to an input sequence of length n , *execManager*, returns a pair of the final $(n + 1)^{th}$ state and the sequence of external output (in the implementation description y) and implicit output (in the implementation description x , also referred to as state) pairs for all previous n :

```
(x(n), [(x(0),y(0)), (x(1),y(1)), ... , (x(n-1),y(n-1))])
```

where the ellipses indicate the pairs between 1 and $n - 1$ ($n \geq 4$).

An example use is:

```
execManager (\x u -> (u,2*x), 9) [1,3,7]
```

which returns

```
(7, [(9,18), (1,2), (3,6)])
```

So, running a system of a delay with output scaled by two and initial state 9 on the input sequence [1,3,7] gives final state 7, outputs [18,2,6], and evolves through the intermediate states [9,1,3].

The execution manager can now be used to execute the block diagram in Fig. 2. The input sequence [3,5,9] is applied by the command

```
execManager exampleBlockDiagram [3,5,9]
```

The result of this execution is

```
(
  (1,10,[1,4,6]),
  [
    ((1, 1,[ ]),1),
    ((1, 4,[1 ]),4),
    ((1, 6,[1, 4]),6)
  ]
)
```

This output has the form (finalState, listOfStateAndOutputPairs), which in this case is

```
(
  x(3),
  [
    (x(0), y(0)),
    (x(1), y(1)),
    (x(2), y(2))
  ]
)
```

where the state, x has the form (ConstantState, DelayState, RecorderState). The final output (1,10,[1,4,6]) then corresponds to (ConstantState = 1, DelayState = 10, RecorderState = [1,4,6]), as expected. Note that the *Delay* has an initial state value of 1.

IV. TIME IN SINGLE-RATE EXECUTION

So far, the block diagram execution as obtained by the execution manager in Definition 6 is atemporal. It is a repeated application of θ . Time has not been part of the definitions as the block diagram was assumed to be of single rate. All states advanced simultaneously, and so a bare sequence of states ensues.

For the single rate block diagram models, a straightforward temporal interpretation is allowed by assuming periodicity of evaluations. In other words, there is a constant interval of time, $h = t_{i+1} - t_i$, between each of the repeated evaluations θ . To implement this, the execution can be redefined as a repeated application of $\{\theta(\psi, z, t_i) \circ (t_{i+1} = t_i + h)\}$. The sequence of tuples that comprise the trajectory in Definition 7 are then equally spaced in time.

Definition 8 (Temporal Behavior): A temporal behavior is a sequence of triples $\langle Y, X, t \rangle_i$ with $i \in [1..\mu]$, $t_{i+1} - t_i = h$ and $\mu \in \mathbb{N}$ and $h \in \mathbb{R}$.

V. CONTINUOUS-TIME SYSTEMS

While embedded controller models are well captured by execution in discrete-time, plant models are typically designed using ordinary differential equations (ODE) or differential and algebraic equations (DAE) [2]. This requires the execution to handle continuous-time behavior evolution.

A. Representing Continuous-Time

The Integrator block in a block diagram represents continuous states and models the time-integration operation $x(t_{i+1}) = x(t_i) + \int_{t_i}^{t_{i+1}} u(t)dt$, with $u(t)$ the input to the Integrator. The output, $y(t_i)$, of the Integrator at time t_i equals the stored state, $x(t_i)$, at that point in time.

Numerical simulation of continuous-time systems typically proceeds by computing a finite number of points on the continuous curve. This results in a representation that is discretized in time. There are, however, some critical differences between discrete-time and continuous-time blocks. In particular, the new state computation in discrete-time can be a trivial mapping $x_i(t_{i+1}) = z_i(t_i)$ whereas in continuous-time the state at the next point in time, $x(t_{i+1})$, has to be *inferred* from its gradient in time at one or more preceding points in time, $\dot{x}(t_i), \dot{x}(t_{i-1}), \dots, \dot{x}(t_0)$, with t_0 the initial time at which execution of the block diagram starts.

This inferencing scheme is often referred to as the *numerical solver*, or *solver* for short, and the algorithm it implements can be quite elaborate [12]. What sets apart continuous-time and discrete-time in this context are the continuity constraints of the continuous-time trace that the solver employs. For example, if the solver is based on a Runge-Kutta fourth order algorithm, it assumes the continuous time trace has well-defined time-derivatives up till the fourth order, i.e., the signal is C^4 . This assumption then translates into the type of curve that is being fit between the computed points.

In the definition of a block diagram in Section II, the computations performed by the solver are executed in the function θ , which, therefore, can become quite complex. An additional assumption that now is required is that the variables involved in a block diagram are complex. In addition, any restriction that the solver requires, such as the function it integrates being Lipschitz and having a certain continuity, carry over as a requirement on the block diagram formulation.

B. Computing Continuous-Time Trajectories

The most straightforward integration scheme is a *forward Euler*, which multiplies the time-derivative, $\dot{x}(t_i)$, at a time t_i by the step size, $h = t_{i+1} - t_i$, and adds this to the current state, $x(t_i)$, to produce the state at t_{i+1} as $x(t_{i+1}) = h\dot{x}(t_i) + x(t_i)$.

The forward Euler can be used to transform a hybrid dynamic system to a completely discrete representation. However, because of its simplicity, this approach can give a false sense of appropriately dealing with continuous-time systems. Therefore, in this work a more advanced

second order Runge-Kutta algorithm is implemented. The integration equations are

$$\begin{aligned} m_1 &= h\phi(x(t_i), t_i) \\ m_2 &= h\phi(x(t_i) + 0.5m_1, t_i + 0.5h) \\ x(t_{i+1}) &= x(t_i) + m_2 \end{aligned} \quad (2)$$

The function θ in Definition 6 is then straightforward to derive as

$$\theta : \begin{cases} \langle \psi(t_i), z(t_i) \rangle = \phi(v(t_i), x(t_i), t_i) \\ m_1 = hz(t_i) \\ \langle \psi(t_i + 0.5h), z(t_i + 0.5h) \rangle = \\ \quad \phi(v(t_i + 0.5h), x(t_i) + 0.5m_1, t_i + 0.5h) \\ m_2 = hz(t_i + 0.5h) \\ x(t_{i+1}) = x(t_i) + m_2 \end{cases} \quad (3)$$

where $v \in \Upsilon$, $\psi \in \Psi$, $x \in X$, $z \in Z$, and m_1 and m_2 with the same dimension as x .

Note that computationally, this is still a single-rate system, where the time interval between each consecutive pair of computations is $0.5h$. However, the v , x , and z values that are computed at each $t_i + 0.5h$ in Eq. (3), also referred to as *minor time steps* [14], are not stored.

VI. HYBRID DYNAMIC SYSTEMS

To operate on a block diagram with mixed discrete-time and continuous-time states, the block diagram definition is first extended to distinguish between continuous-time states, X_c , and discrete-time states, X_d , with $X \equiv X_c \cup X_d$ and $X_c \cap X_d \equiv \emptyset$. This definition applies similarly to the Z space.

Definition 9 (Hybrid Block Diagram): The `HybridBlockDiagram` is a tuple $\langle \Upsilon, X_c, X_d, \Psi, Z_c, Z_d, \phi \rangle$ with $\phi : \Upsilon \times X_c \times X_d \rightarrow \Psi \times Z_c \times Z_d$ and $\Upsilon \in (\mathbb{R} \times \mathbb{S})^m$, $X \equiv X_c \cup X_d \in (\mathbb{R} \times \mathbb{S})^n$, $\Psi \in (\mathbb{R} \times \mathbb{S})^l$, $Z \equiv Z_c \cup Z_d \in (\mathbb{R} \times \mathbb{S})^n$, and $l, m, n \in \mathbb{N}$.

A. A Loose Implementation

In a straightforward implementation, it is tempting to simply combine the θ function of the continuous-time and single-rate discrete-time executions. In this approach, the combination of the second order Runge-Kutta integration scheme and the discrete-time state update yields

$$\begin{aligned} \langle \psi(t_i), zc(t_i), zd(t_i) \rangle &= \phi(v(t_i), xc(t_i), xd(t_i), t_i) \\ m_1 &= h \cdot zc \\ \langle \psi(t_i + 0.5h), zc(t_i + 0.5h), zd(t_i + 0.5h) \rangle &= \\ \quad \phi(v(t_i + 0.5h), xc(t_i) + 0.5m_1, xd(t_i + 0.5h), t_i + 0.5h) \\ m_2 &= h \cdot zc(t_i + 0.5h) \\ xc(t_{i+1}) &= xc(t_i) + m_2 \\ xd(t_{i+1}) &= zd(t_i) \end{aligned} \quad (4)$$

where $xd \in X_d$, $xc \in X_c$, $zd \in Z_d$, $zc \in Z_c$, $v \in \Upsilon$, $\psi \in \Psi$, and m_1 and m_2 with the same dimensions as xc .

However, even if the step size h is chosen such that the computed time points, t_i , coincide with the sample times of the discrete-time states, this is not a single-rate system. The continuous-time computations execute at a rate of $0.5h$, while the discrete-time computations execute at a rate of h . This problem cannot be solved by simply executing the discrete-time computations at a faster rate,

i.e., at $0.5h$ because the values of the continuous-time computations are not available at each $0.5h$ interval. To obtain those values, the continuous-time computations need to be executed at a time interval of $0.25h$, using the Runge-Kutta integration scheme in Eq. (3). So, inevitably, a multi-rate system ensues.

In a less strict approach, this complication can be circumvented by observing that, if the discrete-time execution is performed at a rate of h , the values of the discrete state do not change between t_i and $t_i + 0.5h$, and, therefore, the continuous-time computations are consistent and correct, even though the execution rates do not match.

An example block diagram that embodies such a multi-rate behavior is presented in Fig. 3. The source block *Ramp* provides an increasing input value with a constant slope (time-derivative). This input is delayed by 1 [s] by the *Delay* block before it is input to a continuous-time *Integrator*. The output of this integrator is used as input to the *Gain* block that executes at a sample rate of 1 [s] as well. The output of the *Gain* block is made available by the output port *Out*, for example, for display purposes.

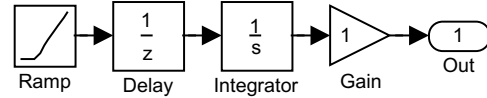


Fig. 3. A hybrid dynamic system with implicit rate transitions.

The *Delay* block computes its output every 1 [s], and so to implement the Runge-Kutta numerical integration algorithm, the *Integrator* can perform a computation with step size $h = 2$. This implies that the *Gain* could not execute with the sample rate of 1 [s], though.

If the *Integrator* is to have an output available every 1 [s], it should be evaluated each 0.5 [s], but the *Delay* only executes each 1 [s]. At this point, it should be clear that a rate transition needs to be inserted. However, in a loose implementation it can be considered that the output of the *Delay* does not change in between its sample times, and, therefore, the *Integrator* can use the *Delay* output at a rate of 0.5 [s].

This loose implementation is brittle and prone to failure. In addition, it allows an incorrect mental model of the model designer with respect to rate transitions, and this can make it difficult to convey the necessity of rate transitions in multi-rate systems where the argument of a non-changing discrete value does not hold.

B. A Strict Implementation

In a strict implementation, an explicit rate transition needs to be inserted between the continuous-time computations and the discrete-time computations.

There are two possible directions of the rate transitions:

- From discrete time to continuous time. In this case, often a zero-order hold is applied, which is the implicit transition in the loose implementation. However, any

other scheme to interpolate between the bounding discrete-time values can be chosen.

- From continuous time to discrete time. In this case, the continuous-time signal is sampled at the points in time at which the discrete-time value is required.

To implement the discrete-time to continuous-time rate transition, a state variable has to be introduced. For a zero-order hold implementation, this state variable changes at the slow rate, but can be read at the high rate. In other work [16], this state behavior is achieved by replacing block diagram connections in Simulink by data stores in a data flow model, where the connection read and write operations are honored.

The continuous-time to discrete-time rate transition has an important implication. It requires the numerical integration algorithm, such as the Runge-Kutta, to employ a step size of which the discrete-time sample rate is an integer multiple. Otherwise, no knowledge of the continuous-time value is available when it needs to be sampled into a discrete-time representation.

The example in Fig. 3 is modified accordingly in Fig. 4. Two rate transition blocks have been inserted, one between the *Delay* and the *Integrator* and one between the *Integrator* and the *Gain*.

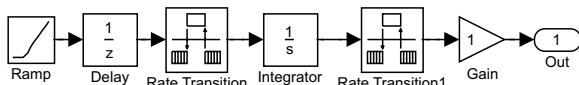


Fig. 4. A hybrid dynamic system with explicit rate transitions.

VII. CONCLUSIONS

The interaction between the continuous-time and discrete-time behavior of embedded control system models is the focus of this paper. It has been shown how there are subtle intricacies when combining the two domains in an execution environment.

In high-level languages, these subtleties are often resolved automatically. This prevents the modeler from getting bogged down in the details of semantics that on many occasions may not be of interest. As such, a loose semantics boosts productivity and increases the user friendliness of a modeling environment.

The apparent drawback is that users may not be aware of what is implicitly implemented. As a result, they may not be aware that there could be alternatives to the chosen implicit implementation. On the occasion where these alternatives could be more appropriate, the user does not have the astuteness to modify the implicit implementation.

Of course, this should not be misconstrued as an argument to confront the user with any low-level implementation detail. The challenge is to choose the correct set of details to hide in the implicit implementations, which is then the default behavior.

The unveiling of implicit assumptions also is of great benefit in the language and tool design. For example, though

in the high-level, and loosely defined, language, explicit rate transition between discrete time and continuous time may not be required, the awareness that it is implicitly implemented leads to the observation that the output of blocks on the cusp of discrete time and continuous time are truly state variables. This is a critical observation to be able to correctly initialize a block diagram, and, therefore, important to be understood by the tool designers.

REFERENCES

- [1] A. Balluchi, M. Di Benedetto, C. Pinello, C. Rossi, and A. Sangiovanni-Vincentelli. Hybrid control for automotive engine management: The cut-off case. In T.A. Henzinger and S. Sastry, editors, *Lecture Notes in Computer Science, Hybrid Systems: Computation and Control*, pages 13–32, Berlin, 1998. Springer-Verlag.
- [2] F.E. Cellier, H. Elmqvist, and M. Otter. Modelling from physical principles. In W.S. Levine, editor, *The Control Handbook*, pages 99–107. CRC Press, Boca Raton, FL, 1996.
- [3] Ben Denckla and Pieter J. Mosterman. An intermediate representation and its application to the analysis of block diagram execution. In *Proceedings of the 2004 Summer Computer Simulation Conference (SCSC'04)*, San Jose, CA, July 2004.
- [4] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. Reuse of software in distributed embedded automotive systems. In *EMSOFT'04*, pages 203–210, Pisa, Italy, September 2004.
- [5] Rick A. Hyde. Fostering innovation in design and reducing implementation costs by using graphical tools for functional specification. In *Proceedings of the AIAA Modeling and Simulation Technologies Conference*, Monterey, CA, August 2002.
- [6] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, April 2003. ISBN-10: 0521826144.
- [7] Pieter J. Mosterman and John E. Ciolfi. Interleaved execution to resolve cyclic dependencies in time-based block diagrams. In *Proceedings of the 43rd IEEE Conference on Decision and Control (CDC'04)*, Paradise Island, Bahamas, December 2004.
- [8] Pieter J. Mosterman, Janos Sztipanovits, and Sebastian Engell. Computer automated multi-paradigm modeling in control systems technology. *IEEE Transactions on Control System Technology*, 12(2), March 2004.
- [9] Klaus D. Müller-Glaser, Gerd Frick, Eric Sax, and Markus Kühl. Multi-paradigm modeling in embedded systems design. *IEEE Transactions on Control System Technology*, 12(2), March 2004.
- [10] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, Hoboken, NJ, 1992. ISBN 0 471 92980 8.
- [11] Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Lecture Notes in Computer Science*, volume 2562, pages 376–390, New Orleans, LA, January 2003. Springer-Verlag. Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages.
- [12] Linda R. Petzold. A description of DASSL: A differential/algebraic system solver. Technical Report SAND82-8637, Sandia National Laboratories, Livermore, California, 1982.
- [13] Ernesto Posse, Juan de Lara, and Hans Vangheluwe. Processing causal block diagrams with graph-grammars in atom3. In *Proceedings of the European Joint Conference on Theory and Practice of Software (ETAPS)*, pages 23 – 34, Grenoble, France, April 2002. Workshop on Applied Graph Transformation (AGT).
- [14] Simulink. *Using Simulink*. The MathWorks, Natick, MA, 2004.
- [15] Frits W. Vaandrager and Jan H. van Schuppen, editors. *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1999.
- [16] Takanori Yokoyama. An aspect-oriented development method for embedded control systems with time-triggered and event-triggered processing. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2005)*, San Francisco, CA, March 2005.