Stream- and State-Based Semantics of Hierarchy in Block Diagrams

Ben Denckla * **Pieter J. Mosterman** **

* *Denckla Consulting, USA (Tel: 310-275-4249; e-mail: bdenckla@alum.mit.edu).*
** *The MathWorks, Inc., Natick, MA 01760 USA (e-mail: pieter.mosterman@mathworks.com)*

Abstract: Block diagrams are often used in embedded system design for modeling both plant and controller, typically with continuous and discrete modeling, respectively. Though easy to use, advanced users and implementers of these languages often run afoul of subtle semantic problems these seemingly simple languages can have. Based on the stream- and state-based approaches, this paper discusses how the specialized state-based semantics of continuous-time block diagrams can interoperate hierarchichally with discrete-time block diagrams. The languages presented may serve as a reference of sorts, helping to clarify some of the underlying choices in block diagram language design, and in the process shedding light on the differences between, and limitations of, existing block diagram languages.

1. INTRODUCTION

To manage the complexity of engineered systems three concepts can be applied: (i) partitioning, (ii) modularity, and (iii) hierarchy. This work adds the notion of hierarchy to previous work by Denckla and Mosterman [2004, 2005] and Denckla et al. [2005] on developing a precise semantics for block diagrams. A block diagram consists of a collection of blocks with geometric shapes that represent mathematical operations on a set of input variables to produce output variables. Such diagrams are also referred to as *causal block diagrams* by Posse et al. [2002], *hierarchical signal-flow diagrams* by Karsai et al. [2003] and *time-based block diagrams* by Mosterman and Ciolfi [2004].

Block diagrams are the basic language of the widely used commercial product Simulink® (Simulink [2004]). Block diagrams support continuous time as well as discrete time, which is important to facilitate analysis of a continuous-time plant model in combination with a discrete-time controller model. The combination of continuous-time and discrete-time models leads to a mathematical representation called a *hybrid dynamic system*.

In this context, the precise definition of how hierarchy is implemented is critical in the context of formalizing block diagrams. In industrial strength design products different implementations of hierarchy are supported. For example, in Simulink hierarchy exists in the form of virtual and nonvirtual subsystems, libraries, referenced models, and referenced binaries with a well-defined application programming interface (API).

Though users desire such flexibility, it complicates a formal analysis of the language. In particular, support for binaries through an API is difficult to analyze as it pertains to a language other than block diagrams. The work presented in this paper seeks to be more precise, and, as such, it will be more limited in its flexibility to facilitate different implementations of hierarchy. In another sense, however, this work is more flexible as it will analyze the use of hierarchy in the context of both stream-based execution as well as state-based execution.

A stream-based approach to analyzing block diagrams is a natural fit that allows a functional perspective on the block diagram language. As such, it supports developing precise and unambiguous semantics, which has been exploited in work by Halbwachs et al. [1991], Halbwachs [1998] and Halbwachs and Raymond [1999] to design a language for safety critical systems. This language applies to discrete-time reactive systems only, though.

This work investigates the differences between stream-based and state-based execution and studies their use in hierarchical hybrid dynamic system decomposition. The state-based approach is presented as more amenable to handling continuous-time behaviors because it requires sophisticated state-based numerical integration algorithms.

In other work, Caspi and Pouzet [1997] examine the differences between stream- and state-based discrete system models outside the block diagram context. It covers the important issue of the limited scope of the state-based model. For example, the state-based model cannot easily describe multirate systems, whereas the stream-based model can.

In work by Liu et al. [2004] the state-based approach is effectively used to model hybrid dynamic systems, where a *director* implements the integration algorithms. The block diagram language is then specified by a mapping onto an imperative programming language. Instead, this work takes a higher level approach to language design by using *lambda calculus* (Nielson and Nielson [1992]) as a general model of computation onto which the block diagram language is mapped. This eliminates the need of introducing imperative concepts at an early state in the language design, and, therefore, reasoning on the semantics can be at a higher level and without the need to deal with the specific execution details.

Related work by Lee and Sangiovanni-Vincentelli [1998] uses a denotational approach as well but for the objective of analysis, rather than providing an executable form. The work presented here uses the declarative language Haskell (Jones [2003]) to provide an executable representation of lambda calculus, so that, for example, reference behaviors can be generated to compare with more efficient implementations of block diagram simulators such as Simulink. Note that Haskell has been used to define noncausal models as well Nilsson et al. [2003].

Another approach to language design by de Lara et al. [2004] employs meta-modeling in combination with graph grammars. The concrete and abstract syntax with its well-formedness constraints are modeled with class diagrams or entity-relationship diagrams. The semantics is then modeled in either: (i) a denotational sense, by translating a model in the newly designed formalism to a corresponding model in an existing formalism; or (ii) an operational sense, by modeling how the state of the model changes. This approach could well be complementary to the one presented in this paper, provided that it would be adapted to handling textual languages as well, in which case the graph grammar may degenerate to a tree grammar.

Section 2 first presents the stream-based and state-based forms of execution. Section 3 introduces the notion of hierarchy, in particular in the context of block diagrams. Section 4 uses hierarchy to support the embedding of the state-based execution in a stream-based execution, which supports the design of a digital controller based on a continuous-time plant model. Section 5 presents conclusions of this work.

2. TWO FORMS OF EXECUTION

Signal processing engineers often apply an input/output view on a system. The system takes a stream of input data and produces a stream of output data. Control system engineers, on the other hand, often apply a state-based view. In this view, the system is considered to be in a certain state, and only the present input is applied to compute the present output.

2.1 Two Block Diagram Semantics

These different approaches are discussed and a semantics for each of them is developed based on the abstract syntax of block diagrams. The semantics of two closely related block diagram languages are discussed, BDFUN and BDSYS that share their syntax but differ in their semantics; BDFUN is stream-based and BDSYS is state-based.

The semantics of BDFUN and BDSYS can be given as a relatively simple translation to any language with lazy evaluation (i.e., expressions are only evaluated when necessary) and a recursive 'let' construct (i.e., equations in a system of equations may be mutally dependent). The premier such language is Haskell (Jones [2003]).

The translation from BDFUN to Haskell proceeds generally as follows. Give a unique identifier (ID) to each arrow tail junction. An arrow tail junction (ATJ) is a point where one or more arrow tails coincide. Translate each block to a declaration of the form $y = f\ u$ where

Table 1. General Haskell block diagram form

$$
\begin{array}{l}
\lambda(i, i, \ldots) \rightarrow \\
\quad \text{let} \\
\qquad (i, i, \ldots) = e(i, i, \ldots) \\
\qquad (i, i, \ldots) = e(i, i, \ldots) \\
\qquad \ldots \\
\quad \text{in } (i, i, \ldots)
\end{array}
$$

- y is a tuple of the IDs of the ATJs on the block,
- u is a (possibly empty) tuple of the IDs of the ATJs for each arrow whose head is on the block, and
- f is the translation of the BDFUN expression inside the block.

Translate the diagram to

- a λ expression binding a (possibly empty) tuple of the IDs of the ATJs that are not on any block, where the body of this λ expression is
  - a **let** expression containing the (possibly empty) list of declarations from the block translations, where the body of this **let** expression is
    a tuple of the IDs of the ATJs for each arrow whose head is not on any block.

So, in general block diagrams translate to expressions of the form shown in Table 1, where i is a meta-variable ranging over the IDs of the ATJs and e is a meta-variable ranging over the expressions that result from translating the contents of the blocks.

Rather than presenting the general translation in detail (see Denckla et al. [2005]), translations of BDSYS into Haskell will be given on a case-by-case basis.

2.2 Stream-Based Execution

In BDFUN, blocks and block diagrams are functions whose input and output are streams. In this work, a stream consists of an infinite sequence of values. Unlike in work by Lee and Sangiovanni-Vincentelli [1998], a tag is not associated with each value, since the position in the sequence is sufficient to capture the behavior of the block diagrams considered here.

A conventional recursive definition of streams is used: if a stream of values of type A is denoted $S\ A$, then $S\ A = (A, S\ A)$. Or, a stream of values of type A is a pair of value of type A and a stream of values of type A.

A block (or a block diagram) is defined to be a function $S\ A \rightarrow S\ B$ for some types A and B. In other words, a block is a function taking a stream of A values as input and returning a stream of B values as output.

To illustrate, the first-order discrete-time system in Fig. 1 is translated into a Haskell function, for which the definitions of Table 2 are used. The function *uncurry* converts a curried function to a function on pairs, *curry f x y = f (x, y)*. The operator *map* applies a function across a list of values, *map f (x:xs) = f x : map f xs*. The operator *zipWith* applies a binary function across two lists of values, *zipWith z (a:as) (b:bs) = z a b : zipWith z as bs*.

Table 3 shows Fig. 1 translated into a Haskell function using BDFUN semantics and the definitions of Table 2. The functions *adder* and *gain* are 'streamifications' of

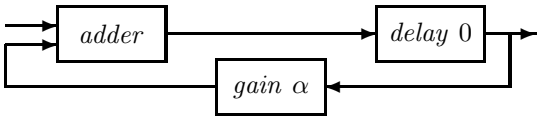


Fig. 1. A first-order discrete-time system

Table 2. Definitions for translating Fig. 1

```
delay xi  = (xi :)
gain α = map (α*)
adder = uncurry (zipWith (+))
```

Table 3. BDFUN translation of Figure 1

```
let
    f u = y
        where
        y = d
        d = delay 0 a
        g = gain α d
        a = adder (u, g)
in f
```

Table 4. Definitions for translating Fig. 2

```
delay xi  = (f, xi) where f (x, u) = (u, x)
adder = uncurry (+)
gain α = (α*)
```

their scalar counterparts: For a given set of synchronous elements in the input and output streams, the output of the *adder* block is the sum of its inputs. The output of the *gain* block is its input multiplied by the factor α. However, *delay* has no scalar counterpart. The expression *delay xi* prepends *xi* to its input stream. The colon can be thought of as the 'prepend' operator in Haskell.

2.3 State-Based Execution

In BDSYS, a block (or a block diagram) is a pair (f, x) where f is a function and x is an initial state.

The function f has type $X \times U \to Z \times Y$ where $x \in X$, U is the input type, Y is block's output type, and Z is the "implicit output" type of the block. For discrete systems, the implicit output is the next state, so $X = Z$. For continuous systems, the implicit output is the derivative of the current state. It is a matter of preference whether in the continuous case X is considered equal to Z. For example it may be desirable for X to have type "meters" and Z to have type "meters per second."

The Semantics Defined A block in a BDSYS diagram does not have its state input or implicit output shown because they are always treated the same. That is why the concrete syntax is identical to that of a BDFUN diagram.

The BDSYS semantics can be applied to the example in Fig. 1. For clarity, the state input *dx* and implicit output *dz* are shown in Fig. 2, though they would not normally be shown. To translate this system into Haskell, the definitions of Table 4 are used. Note that the *delay* function takes an initial state and returns a system *(df,dxi)*.

Table 5 shows the translation of Fig. 2 into a Haskell function using BDSYS semantics. The functions *adder* and

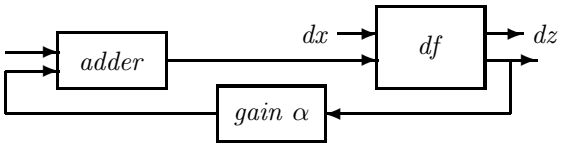


Fig. 2. Function part of a 1st-order discrete system with state input and implicit output

Table 5. BDSYS translation of Figure 1

```
let
    (df, dxi) = delay 0
    xi = dxi
    f (dx, u) = (dz, y)
        where
        y = d
        (dz, d) = df (dx, a)
        g = gain α d
        a = adder (u, g)
in (f, xi)
```

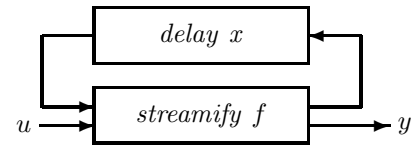


Fig. 3. Discrete execution manager core

gain are scalar, and neither take nor produce state. The function f returned by *delay* is more interesting. Given (x, u) (a current state x and an input u), it returns (u, x), a next state of u and an output of x.

The Execution Manager The system produced by BDSYS can only be run on a single input as opposed to a stream of inputs. Such systems can be run on a stream with the help of an *execution manager*. Given a system (f, x) and a stream of inputs u, an execution manager applies f to each element of u while evolving the state forward from x. In particular, a discrete-time execution manager produces a stream of states z and outputs y as follows:

$$\begin{aligned}(z_1, y_1) &= f(x, u_1)\\ (z_2, y_2) &= f(z_1, u_2)\\ (z_3, y_3) &= f(z_2, u_3)\\ &\dots\end{aligned}$$

Note how the state output of one computation, e.g. z_1, feeds back as the state input of the next computation.

More formally, an execution manager is a function $F \times X \to (S\ U \to S\ Y)$ where $F = X \times U \to Z \times Y$. In other words an execution manager is a function taking an (f, x) pair and returning a stream-transforming function.

The core work of a discrete execution manager can be shown in a BDFUN block diagram such as Fig. 3. This diagram shows the stream version of the system (f, x). (It assumes a *streamify* function is available, which is just the *map* function surrounded by appropriate conversions between pairs of streams and streams of pairs.)

Table 6. Expression that gives meaning to Fig. 1

```
let
    delay xi  = (xi :)
    adder = uncurry (zipWith (+))
    gain α = map (α∗)
in d
```

3. HIERARCHY

Often, hierarchy in block diagrams is introduced by defining a *primitive* block in terms of some language with executable semantics and a *compound* block by an interconnection of blocks (which can be primitive and compound). Much like in work by Reekie [94], here compound and primitive blocks need not be distinguished as the block diagram language and Haskell can be used interchangeably.

3.1 Block Diagrams as an Extension to Haskell

In Fig. 1 and Fig. 2, the meaning of the blocks is given by what is written inside of them. The translation assumes that the text inside a block, e.g. *adder*, is a Haskell expression. This is a significant design choice that is easy to overlook. This section examines some of the implications of the choice of a language for use inside blocks. The text inside of the blocks corresponds to Haskell code, and, therefore, can be executed. This provides a level of hierarchy where Haskell functionality is part of the block diagram. For example, the *adder* block utilizes the Haskell *adder* function. All that is required is to further provide the arguments to the Haskell functionality which can be derived based on the input and output of the blocks.

This implementation allows for viewing BDFUN and BDSYS not as languages in their own right but as extensions to a general purpose 'host' language.

Both the host language and the block diagram languages BDFUN and BDSYS benefit from this combination. The host language is extended by block diagrams and, at the same time, the block diagram language is extended by a language that can be used inside blocks and outside block diagrams. This provides the host language with a syntax that succinctly expresses a generalized form of function composition, as embodied by block diagrams. For example, a connection from block a to block b in a block diagram can be interpreted as the function composition $b \circ a$.

To illustrate, consider the block diagram in Fig. 1. If block diagrams are viewed as a syntactic extension of Haskell, the expression in Table 6 gives full meaning to Fig. 1 by resolving its external references, *adder*, *delay*, and *gain*.

Note that all the functions used in defining *adder*, *delay*, and *gain* are built into Haskell. Now Fig. 1 can be executed. Assuming that the function that is expressed by Fig. 1 is assigned to the variable *dr* and $\alpha = 0.5$, then *dr* $[1, 0, 0]$ gives result $[0.0, 1.0, 0.5, 0.25]$.

3.2 The Language Inside Blocks

Hierarchy is introduced by the language inside the blocks. The simplest language inside a block just allows a block to reference one of a predefined set of built-in functions. A next step may be to allow a block to call one of a predefined set of built-in function constructors, i.e., functions that return functions. For example a gain of 2 might be expressed by the text '*gain* 2' inside the block. Two types of hierarchy beyond this, nesting and referencing, are discussed next.

Nesting Hierarchy If a block can be defined not just by text inside it but also by a block diagram inside it, then the language can be said to support *nesting hierarchy* or just *nesting*. Nesting is analogous to introducing a local variable and only using it once. For example,

$$\text{let } x = 2 * 3 \text{ in } x * 4$$

could be said to hide $2 * 3$ underneath x in $x * 4$ in the same way nesting may be used to take a block diagram and hide it underneath a block.

An example of block diagram nesting is given in Fig. 4. Here, the language in the left-most block is the block diagram language.

Referencing Hierarchy Another type of hierarchy allows a block to reference a function defined outside the block diagram. For example, a block may reference a function f whose definition, $f\ x = x + 3$, is outside the block diagram. If a block can reference a function defined by a block diagram, then the language can be said to support *reference hierarchy* or just *reference*. Reference in a block diagram is analogous to the use of an unbound variable in an expression, for example, x in $x * 4$.

Conclusions Ideally, if d is a subdiagram of D, the meaning of D is unchanged if d is replaced by a block that nests or references d. Instead of defining 'subdiagram' here; it is noted that it is analogous to a subexpression, e.g. $(2 + 3)$ is a subexpression of $(2 + 3) * 5$.

Endowing a language with these forms of equivalences under substitution has long been acknowledged as a goal of language design (Landin [1966]), yet is rarely achieved in practice. In particular, in the case where semantics are not precisely defined, such equivalence is easily violated. It is the aim of this work to be formal and consistent with commonly accepted language desiderata.

4. EMBEDDING OF STATE-BASED INTO STREAM-BASED EXECUTION

Intricacies of continuous-time systems make state-based execution the most suitable approach. Hierarchical decomposition embeds a continuous-time system into a stream-based execution.

4.1 Continuous-Time Systems

To simplify matters, the execution of a continuous-time system is first considered where the system takes no input or output (i.e., a system whose function part has the type $X \to Z$). This is called a *standalone system*.

Because continuous-time dynamics require integration with respect to time to generate a behavior, a continuous-time system can be more complex to execute than a discrete-time system. A *solver* performs this integration

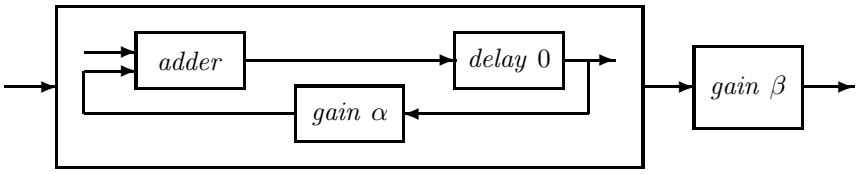


Fig. 4. A block diagram hierarchy

Table 7. BDSYS translation of Figure 5

```
let
    (if, ixi) = ∫
    xi = ixi
    f (ix, u) = (iz, y)
        where
        y = i
        s = sub (u, i)
        g = gain 1/rc s
        (iz, i) = if (ix, g)
in (f, xi)
```

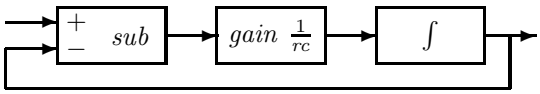


Fig. 5. A continuous-time system

by providing a block diagram with a state and operating on the derivatives with respect to time that are output by the block diagram.

This implies that the block diagram has to allow the state to be input and it should output the derivatives with respect to time of the state. Because the stream-based approach does not allow for the explicit manipulation of the block diagram state, it is not well suited for executing continuous-time systems. The state-based approach is then most suitable for executing continuous-time systems.

Because in continuous time the implicit output is interpreted as the derivative of the state of the system, different symbols have been used for the types of state and implicit output. Although Z always has the same dimensions as X, it has different units than X and thus may be considered to be of a different type.

Figure 5 depicts an example of a continuous-time system, a first-order lowpass filter with the Haskell code given in Table 7. The expression is a system (f, xi) where three equations hold. The first just decomposes the system indicated by the integral sign into its function and initial state parts (if, ixi). The second equation says that the initial state of the overall system, xi, is just the initial state of the integral system, ixi, since it is the only system in the diagram that has state. The third equation says that f is a function taking a pair (ix, u) as input and returning a pair (iz, y) as output where four equations hold. These four equations are as indicated by a BDFUN interpretation of the block diagram, with state input and implicit output added and the integral replaced by its function part, if.

Given a continuous-time system, s, and a stop time, the solver produces a discrete-time system, σ. The system σ is a version of s which, instead of giving the derivative of the current state, approximates a next state at some time not later than the stop time. Assuming well-behaved dynamics, the execution manager then iterates σ until the stop time is reached, producing a sequence of state/time pairs. Note that this execution scheme accommodates fixed-step as well as variable-step solvers. Further note that it is straightforward to extend the execution manager above to systems with explicit output.

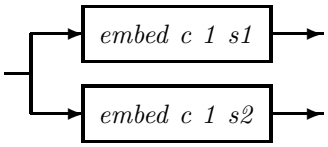


Fig. 6. a hybrid system

4.2 Hybrid Dynamic Systems

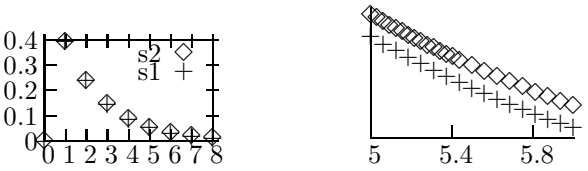
Here, a system is considered to be hybrid if it is a discrete-time system with one or more continuous-time systems embedded in it. The strategy for the embedding is to convert the continuous-time system into a discrete-time system. This 'discretization' is achieved by viewing each step of the discrete time execution manager as a full run of the continuous-time execution manager with any input held constant (i.e., zero-order hold, which is a common part of digital-to-analog converters). All computed states in between the start and stop time of a discrete interval are discarded, except the final state as provided by the continuous-time execution manager. The final state is used as the initial state on the next run. The reason why the intermediate points are computed in the continuous-time system is to satisfy the error tolerances as set for the solver.

Note that this methodology can support multiple solvers running at multiple and/or variable rates all under the control of the same discrete-time system.

For example, in Fig. 6, a hybrid system embeds a continuous-time system c into discrete time using two different solvers, *s1* and *s2*. It does so via a function *embed* which, given a continuous-time system, a sample period, and a solver, gives a discretized version of that system. A call to *embed c t s* returns a discrete-time version of the continuous-time system c. This discrete-time system is formed by running the solver s on c from 0 to time t and then discarding all but the initial and final output. Note that in an actual application, depending on the level of detail that is desired, the digital-to-analog (DA) and analog-to-digital (AD) converters may have to be modeled in detail as well; for the sake of simplicity this is not included and, as such, the system will exhibit considerable imaging during DA side and aliasing during AD side.

Assuming that c is the system in Fig. 5, the result of running this system on an impulse is shown in Fig. 7(a). Though both outputs look the same in discrete time, the execution of the two proceeds in a distinctly different

manner. For example, studying the output between time 5 and 6 in Fig. 7(b), it can be seen that one solver increased its time step, while the other keeps it constant throughout that interval. Note that the points are graphically offset from each other in the vertical direction to make the differences in step size clearer.



(a) Entire interval

(b) Detailed view

Fig. 7. Impulse response

Note that the model of time is included in the block diagram, and, therefore, the tagged signal model is not required at this point. A totally ordered sequence of output suffices, while the temporal interpretation can be derived from the included model of time. This differs in conception from work by, for example, Lee and Sangiovanni-Vincentelli [1998] and Reekie [94] and future research will investigate adapting the presented framework to such tagged signal frameworks.

5. CONCLUSIONS

The flexibility of commercially successful block diagram languages makes it difficult to formally define and analyze the exact behavior of the entire language. This work attempts to provide a precise definition of a block diagram language by taking a functional approach based on lambda calculus. The declarative language Haskell is used to specify an executable denotational form. The denotational nature allows for deferring details about the execution of models and helps concentrate on the specific semantics of the block diagram language. The view of block diagrams as an extension to Haskell provides an array of advantages because mechanisms such as type inferencing, hierarchy, and dependency analysis become available through a well-established programming language.

Using the Haskell specification, it is shown how a state-based approach that is more amenable to handling continuous-time system behavior can be integrated in a stream-based context by means of hierarchical decomposition. This decomposition allows coupling a functional view with a systems view so the desired view for both discrete-time as well as continous-time behavior can be employed.

REFERENCES

P. Caspi and M. Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. Technical Report 07, VERIMAG, Oct. 1997.

J. de Lara, H. Vangheluwe, and P. J. Mosterman. Modelling and analysis of traffic networks based on graph transformation. In *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004)*, pp. 120–127, Braunschweig, Germany, Dec. 2004.

B. Denckla and P. J. Mosterman. An intermediate representation and its application to the analysis of block diagram execution. In *Proc. of the 2004 Summer Computer Simulation Conference*, San Jose, CA, July 2004.

B. Denckla and P. J. Mosterman. Formalizing causal block diagrams for modeling a class of hybrid dynamic systems. In *44th IEEE Conf. on Decision and Control*, Seville, Spain, Dec. 2005.

B. Denckla, P. J. Mosterman, and H. Vangheluwe. Towards an executable denotational semantics for causal block diagrams. In *The 5th OOPSLA Workshop on Domain-Specific Modeling*, San Diego, CA, Oct. 2005.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, Sep. 1991.

N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, pp. 1–16, Vancouver B.C., June 1998. LNCS 1427, Springer Verlag.

N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *Asian Computing Science Conference (ASIAN'99)*, Phuket, Thailand, Dec. 1999. LNCS 1742, Springer Verlag.

S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, Apr. 2003.

G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proc. of the IEEE*, 91(1):145–164, Jan. 2003.

P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966. ISSN 0001-0782.

E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, Dec. 1998.

J. Liu, J. Eker, J. W. Janneck, X. Liu, and E. A. Lee. Actor-oriented control system design: A responsible framework perspective. *IEEE Trans. on Control System Technology*, 12(2), Mar. 2004.

P. J. Mosterman and J. E. Ciolfi. Interleaved execution to resolve cyclic dependencies in time-based block diagrams. In *Proc. of the 43rd IEEE Conference on Decision and Control (CDC'04)*, Bahamas, Dec. 2004.

H. Riis Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, Hoboken, NJ, 1992. ISBN 0 471 92980 8.

H. Nilsson, J. Peterson, and P. Hudak. Functional hybrid modeling. In *Lecture Notes in Computer Science*, volume 2562, pp. 376–390, New Orleans, LA, Jan. 2003.

E. Posse, J. de Lara, and H. 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, Apr. 2002.

H. John Reekie. Modelling Asynchronous Streams in Haskell. Technical Report 94.3, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, May 94.

Simulink. *Using Simulink*. The MathWorks, Natick, MA, 2007.