Submitted to the 2004 Summer Computer Simulation Conference
A. G. Bruzzone and E. Williams, eds.

AN INTERMEDIATE REPRESENTATION AND ITS APPLICATION TO THE ANALYSIS OF BLOCK DIAGRAM EXECUTION

Ben Denckla
Digidesign
2001 Junipero Serra Blvd.
Daly City, CA 94014-3886
U.S.A.

Pieter J. Mosterman
The MathWorks, Inc.
3 Apple Hill Dr.
Natick, MA 01760-2098
U.S.A.

ABSTRACT

This paper presents foundations for a functional execution structure that generates dynamic behavior of data processing systems that are of a sampled nature. These systems are often represented by *block diagrams*, i.e., blocks connected by directed lines, where each block represents data transformation. An intermediate representation defines the semantics of these block diagrams that is directed towards a functional software implementation and should support sophisticated modeling patterns such as feedback and hierarchy. Thus, it provides a point of reference in understanding, analyzing, and comparing existing block diagram simulation software.

1 INTRODUCTION

In this paper, dynamic systems are considered that are generally represented by block diagram models. Such systems can be well modeled by time-based block diagrams and sophisticated analysis and synthesis tools such as Simulink® (Simulink 2004) are readily available. Modeling languages can be classified as being *declarative* or *imperative*. The first makes statements about its next state and internal input and output values while the second commands changes to its state and internal input and output values. As such, an imperative language implements the execution of its constructs that, in a declarative language, can still be chosen. For example, many different orders of execution may lead to the same result in a declarative language, providing room for optimization.

Though block diagram models are declarative, they are *causal*, i.e., the functions that describe the behavior of a block have explicit input and output variables. This differs from other work (Nilsson, Peterson, and Hudak 2003) where a functional approach for non-causal models is developed. Such non-causal models are beneficial in, e.g., the domain of plant modeling but undesired in, e.g., the domain of control algorithm design.

In this paper, a functional approach is taken that resembles the computational structure of Simulink. It applies a *function* (in the mathematical sense, as opposed to the software sense) to perform the data transformation as required by each of the blocks in a model. The function, then, is a concise description of the behavior of the block and it lends itself well to be translated into computer code.

The use of functions to describe block behavior is a first step towards implementing an execution environment. Next, these functions need to be called repeatedly to advance behavior. The use of partitioning to manage system complexity (Wijbrans 1993) brings about the need for input and output signals to each block. This may call for a separation of the one function that captures the block behavior into a state-related and an output-related part. It is shown how hierarchy requires that the state-related and output-related parts may have to be interleaved throughout levels of hierarchy to efficiently execute the model.

Section 2 lays down the principles of functions and the function graph used in this paper. Section 3 then presents an execution mechanism for this function graph. In Section 4, pathological cases that result in cyclic dependencies are introduced. Section 5 gives a mapping of block diagrams onto the function graph and shows how different forms of aggregation may induce or avoid cyclic dependencies. In Section 6, the conclusions of this work are presented.

2 The Basis: Functions

A functional description of behavior relies on a function call that takes a set of input arguments and produces output in return. There is no order of evaluation implied in either the set of input or output arguments. A function can model system dynamics by computing the change of the system state.

2.1 The Dynamics of a Function

In sampled data systems, each variable takes a value at discrete time points, k with $k \in \aleph$. This includes the class of sampled data systems with equidistant sample points, kT_s, where T_s, is the sample time. Note that the temporal interpretation is secondary and the index k may be regarded as a step in a generic iteration that has no immediate bearing on time. This implies as well that in case of a temporal interpretation, the points are not necessarily equidistant.

This section first develops a foundation for describing systems by using functions. The corresponding language will then serve as an intermediate representation to model block diagram components that contain more sophisticated semantic primitives for which a number of function primitives are required. This intermediate representation is closer to the execution level, and, therefore, serves to analyze execution particulars.

A function, f, is an operation that maps part of a real space of dimensions m, the function *domain*, onto a real space of dimensions n, $f : \Re^m \rightarrow \Re^n$. No restrictions apply other than that the map is deterministic, i.e., one point in the function domain maps onto one and only one point. For example, for $m = n = 1$, a function f computes a value $b \in \Re$ from a value $a \in \Re$, $b = f(a)$.

In case f represents a sampled data system with no external connections, the input a is interpreted as the system *state*, often denoted x. The output b is interpreted to be the next state and the dynamics of a discrete-time system become $x(k+1) = f(x(k))$, where the argument k to x denotes an index in a sequence of states. This index could be tied to a point in time.

To generate a sequence of states, a control structure (the *execution engine*) applies the function f repeatedly. Each application computes a new state, to which f is then applied. This results in the state sequence $\{x(0),\ x(1),\ x(2),\ \ldots\}$. In this scheme, the execution engine cycles through two distinct stages: (i) application of the function, f, also called the *output* stage, and (ii) increasing the index, k, to apply f to the next state, also called the *update* stage.

2.2 A Graphical Representation

In general, multiple functions, f_i, $i \in \aleph$, may have to be evaluated and, to analyze their dependencies, a graph representation will be used.

For a single function, f, with $m = n = 1$, as discussed so far, the graphical representation is given in Fig. 1. Here the input and output are represented by the vertices marked v while the operation f is represented by the vertex marked o. Note that the v vertices are not indexed as it is in principle irrelevant and unknown whether f produces a next value or a previous value or has still other semantics (e.g., they could simply represent input and output variables), as long as the two v variables are conceptually different.

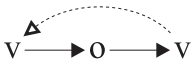


Figure 1: The two stages in the execution engine.

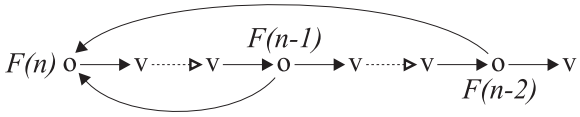


Figure 2: Function graph of the Fibonacci series.

The update stage, though separate from the output computations, can be represented in the same graph by a dashed edge between the two v vertices involved. It ties these vertices together and assigns them 'state' semantics where the origin is the new state, $x(k+1)$, and the destination is the corresponding $x(k)$. Note that the solid and dashed edges are, in general, active during distinctly different stages of the simulation.

A system, S, then is defined by a four-tuple:

$$S =< V_v, V_f, E_o, E_x > \tag{1}$$

where V_v the set of input and output vertices (v vertices), V_f the set of vertices that represent functions (o vertices), E_o the set of edges that constitute the output computations (solid), and E_x the set of edges that constitute the update operations (dashed).

To illustrate the preceding, consider the Fibonacci series

$$F(n) \equiv F(n-1) + F(n-2) \tag{2}$$

A function graph for this is given in Fig. 2. It shows the function $F(n)$ to implement summation of the output of $F(n-1)$ and $F(n-2)$. The shift is achieved by an update edge. Since a regular delay is implemented, the functions $F(n-1)$ and $F(n-2)$ are merely the identity function.

In contrast to other approaches such as object oriented simulation or token exchange execution, in the functional approach presented here, the state variable that is updated on each iteration step is passed into the function, f.

2.3 Aggregating Functions

To manage the complexity of engineered systems, partitioning is applied to model a system as an accumulation of functions. If a number of functions are used to compute the new state from the current state, these can

be combined while expanding the dimensions of the input and output space, and system execution requires only one function call. The input, $a \in \Re^m$ and output $b \in \Re^n$ then become vectors of length m and n, respectively, rather than single values.

Algebraically, the disjoint functions, f_i, that operate on the one-dimensional spaces A_i and B_i are combined into one aggregate function f that operates on the spaces A and B that are the product spaces of the A_i and B_i, $i \in \{1..n\}$, respectively. The operations are completely decoupled, though, i.e., the function f consists of a set of n disjoint operations.

For $n = 2$, the output computation phase of the execution can be graphically depicted, as shown in Fig. 3. In Fig. 3(a) the two separate *o* vertices are shown, where the ellipse indicates they are one aggregate computation. In Fig. 3(b) the separate *o* vertices are collapsed into one supervertex O. Note that the internal, disjoint, structure of the supervertex is lost after the aggregation and dependencies between all input and output appear.

(a) Independent. (b) Aggregate.

Figure 3: Combining functions into an aggregate.

3 EXECUTING SYSTEMS

Given the graph of connected functions, the dependencies can be analyzed to establish an execution order that ensures each of the functions is only evaluated when all its input is available. For example, in Fig. 2, *F(n-1)* only takes exogeneous variables, and, therefore, can be computed first. Similarly, *F(n-2)* is not dependent on output of other functions, and so it can be computed second. Finally, now that *F(n-1)* and *F(n-2)* are known, the dependencies for computing *F(n)* are satisfied, as *F(n-1)* and *F(n-2)* are input to *F(n)* and the execution order becomes $F(n-1) \rightarrow F(n-2) \rightarrow F(n)$.

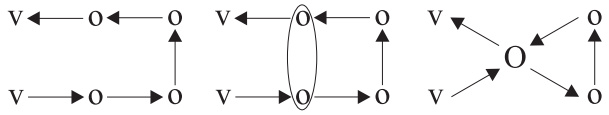
In general, in a declarative approach there are multiple orderings that allow the strict (i.e., with all their input available) evaluation of all functions. Each of these orderings is equally valid and they all produce the same execution results. For the system in Fig. 2, $F(n-2) \rightarrow F(n-1) \rightarrow F(n)$ can be used as the execution order as well, as long as *F(n)* is computed last. Note that, in a denotational framework, the execution order even becomes irrelevant as the result is the fixed point of iterations of randomly executing functions.

4 CIRCULAR DEPENDENCIES

The framework discussed in Section 2 is not *responsible* [adopted from (Liu, Eker, Janneck, Liu, and Lee 2004)], meaning that it allows the design of pathological systems. For example, it is trivial to create a graph with circular dependencies. These cannot be solved by an explicit algorithm that computes function values once and passes its output on, and, therefore, such circular dependencies may have to be eliminated.

Circular dependencies may arise because of aggregating constituent functions into a supervertex. This aggregation may reflect an implementation choice to achieve a certain architecture but the restrictions it places on the execution order may be such that circular dependencies arise. To possibly resolve these dependencies, the graphs that constitute the supervertices involved need to be studied. A different aggregation of the original system may be required to eliminate circular dependencies.

Consider the system in Fig. 4. The original system in Fig. 4(a) contains no circular dependencies. However, when the aggregation in Fig. 4(b) is applied, the circular dependency between the *o* vertices in Fig. 4(c) arises. To remove these, the functions of the original system must be aggregated differently. How fine grained this break-down has to be in case of a hierarchy of supervertices can be optimized (Oberschelp, Gambuzza, Burmester, and Giese 2004).



(a) Initial. (b) Aggregate. (c) Result.

Figure 4: Aggregation may lead to circular dependencies.

5 BLOCK DIAGRAMS

A mapping of a more intuitive and sophisticated modeling formalism onto the intermediate representation is now given. Because of its prevalent use in industry, a specific mapping from time-based block diagrams as found in Simulink is presented.

5.1 Defining Blocks

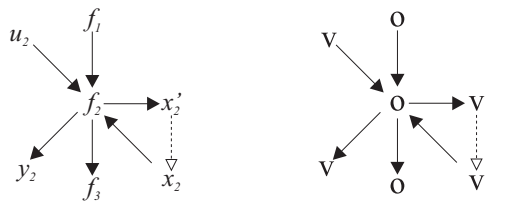
To define blocks for dynamic system simulation, it is observed that they are an accumulation of functions that operate on input and output variables. The in-

put and output of each function distinguishes between state and intermediate variables. This conforms with the function graph introduced in Section 2 where they correspond to v and o vertices, respectively.

To distinguish the respective vertices, the usual block diagram nomenclature will be used in the proceedings, i.e., u indicates input, y indicates output, x indicates the current state, and x' indicates the new state. Functions are indicated by f. Using this notation,

$$f : X \times U \rightarrow X \times Y \tag{3}$$

with X the state space, U the input space, and Y the output space. Figure 5(a) shows how a block may contain a function f_2 to process its input data, u_2 and f_1, based on its state, x_2. The function f_2 computes the block output, y_2, and new state, x_2'. In addition, the block contains a relation between x_2 and x_2' that is executed in the update stage. The corresponding intermediate representation is given in Fig. 5(b).



(a) Annotated. (b) Functional.

Figure 5: A block and its possible connections.

A block then is a system defined by the four-tuple in Eq. (1), with the set V_v being a union of distinguished input vertices, V_u, output vertices, V_y, and state vertices, V_x, i.e., $V_v = V_u \cup V_y \cup V_x$. In the block diagram function graphs, V_u are subscripted u vertices, V_y are subscripted y vertices, V_x subscripted x and x' vertices, and V_f are vertices with an incoming and outgoing solid edge. The different edges, E_o and E_x, and their connected vertices constitute the two graphs that define a block.

In general, an arbitrary number of functions greater than or equal to one can be employed to specify the behavior of a block instead of the one single function f_2 in Fig. 5(a). The output stage of execution may call the functions that constitute a block in any desired order that will result in an explicit sequence of computations. Note that, as such, a block is distinctly different from the supervertex introduced in Section 3.

In the intermediate representation, blocks are then defined by two graphs:

- The *output graph* consists of edges E_o that compute the output variables (x' and y).
- The *update graph* consists of edges E_x that relate current and new states (x and x', respectively).

Note that the current state and new state in the output graph are function input and output, respectively, as well. However, in the context of block diagrams, the differentiation between which input is state and which output is state can be made. The two different graphs will be shown in one depiction, clearly distinguishing between them by using solid edges for the output graph and dashed edges for the update graph. All vertices can be considered part of both graphs (though only the ones connected to an edge are of importance).

5.2 Defining Simulink Blocks

Unlike the general functional implementation of blocks in a block diagram modeling language, Simulink applies a somewhat more specific and restrictive template for sampled data systems and only allows two functions, f and g, to capture the dynamics of a block. The state variables are computed by g, and the intermediate variables by f.

This model of a Simulink block is shown in Fig. 6. In Fig. 6(a), two types of input are possible, distinguished by their *direct feedthrough* character:

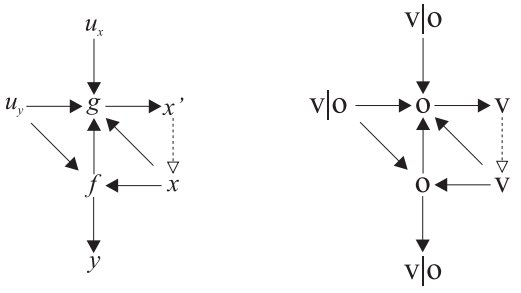
- The input u_x represents input that has no direct feedthrough to the output, i.e., its value is not used in the function f, but it is used only in the function g to compute the new state, x'. This implies that its value does not have to be available when the output, y, is computed.
- The input u_y represents input that does have direct feedthrough. Therefore, its value needs to be available when the block output is computed.

In general, the input, $u = \{u_x, u_y\}$ and the output, y, can be intermediate variables computed by another function (o) or sinks and sources (v). The functional graph of a Simulink block, shown in Fig. 6(b), represents this by indicating $v|o$ instead of either v or o, i.e., both v and o are allowed.

The mapping onto the intermediate representation allows analyses of a block diagram model to identify circular dependencies, called *algebraic loops*, that become immediately apparent in the intermediate representation.

5.3 Connecting Blocks

Given the block definitions using the intermediate representation, a Simulink block diagram can be transformed into a function graph and then be analyzed and optimized.



(a) Annotated. (b) Functional.

Figure 6: Constituents of a Simulink block.

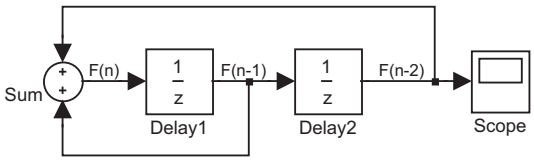


Figure 7: Simulink diagram of the Fibonacci series.

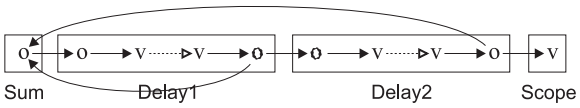


Figure 8: Mapping a block diagram onto functions.

For example, consider the Simulink diagram of the Fibonacci series given in Fig. 7. For the unit delay blocks, *Delay1* and *Delay2*, the f and g functions in Fig. 6(a) are the identity function. Furthermore, there is no direct feedthrough input $u_y = \emptyset$, and the new state, x', is not dependent upon the current state, x. For the sum block, *Sum*, only a function f is present with u_y input. The scope block, *Scope*, is represented by an output variable, or sink, only.

The resulting intermediate representation is given in Fig. 8, which is close to the reduced form given earlier in Fig. 2. The intermediate representation clearly shows that no circular dependencies exist because the update stage is separate from the output stage.

5.4 Aggregating Blocks

In one form of hierarchy, a new block can be defined in terms of existing blocks. In Simulink, a block that contains a number of blocks that define its operation is called a *subsystem*. For example, consider the Simulink

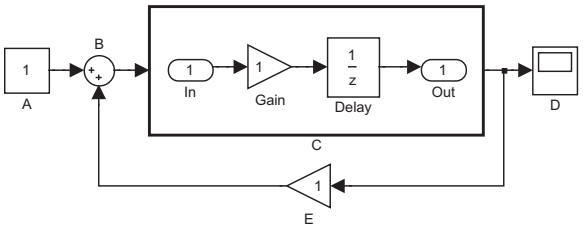
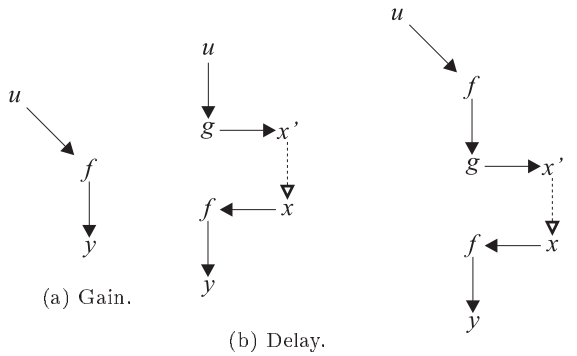


Figure 9: Block diagram with potential algebraic loop.

model in Fig. 9. Here, a new block, C, is defined in terms of its constituents, the gain, *Gain*, and unit delay, *Delay*, while the input and ouput port, *In* and *Out*, respectively, define the interface. To ensure 'block-like' behavior, the subsystem C is marked to be *atomic*, which means that the new block must conform to the template in Fig. 6(a).

Analysis of this new block, C, in terms of the function graph reveals an important potential for conflicts. In Fig. 10(a) first the underlying graph of the gain block is given and in Fig. 10(b) the underlying graph of the unit delay block is given. If these two are aggregated, the graph in Fig. 10(c) emerges.

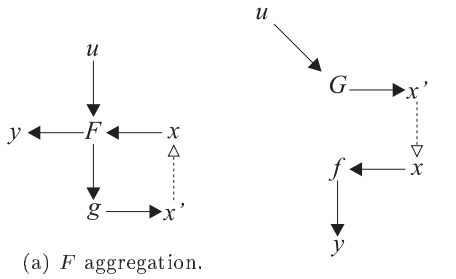


(a) Gain. (b) Delay. (c) Gain/Delay.

Figure 10: Creating a new Simulink block.

To arrive at the canonical form in Fig. 6(a), Simulink aggregates the two f functions into one new function, F, as shown by the supervertex in Fig. 11(a). Instead of the aggregation based on the original interpretation of the meaning of a function, which leads to the aggregate block model in Fig. 11(a), appropriate aggregation would have recognized the change of status of f and move it from the block output computation into the new state computation, as shown in Fig. 11(b).

In case f is not moved to g, Fig. 12 illustrates that an algebraic loop may emerge. In Fig 12(a) the underlying functions of each of the blocks are shown, whereby for C the graph from Fig. 11(a) is used. The constant, A only

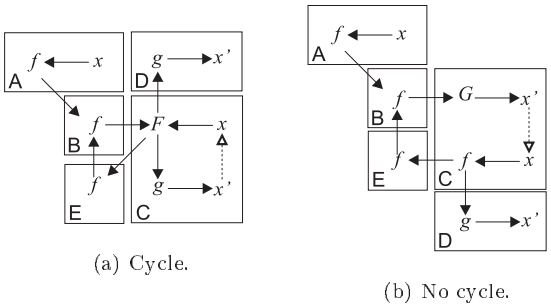


(a) F aggregation.

(b) G aggregation.

Figure 11: Gain/Delay aggregation.

maintains the state part of the general Simulink block in Fig. 6(a). Given that the state is not changed, this produces a constant value. The sum block, B, is given by just the f part with two input ports, much like the gain, E, though it has only one input port. The scope D is modeled as computed state output, x', through an identity function. The intermediate representation clearly shows there is a circular dependency between B, C, and E, in the system.



(a) Cycle.

(b) No cycle.

Figure 12: Use of an aggregate Simulink block.

To eliminate this problem, Simulink can move the f call that is connected to g in Fig. 10(c) into the state computation function, g, of the new block, as shown in Fig. 11(b) (Mosterman and Ciolfi 2004). The result of this is that the cycle in Fig. 12(a) is removed, as shown in Fig. 12(b).

6 CONCLUSIONS

Block diagrams are a proliferated formalism used for modeling engineered systems. To analyze and support the step from block diagrams to a functional description in software, in this paper an intermediate representation is developed. This function graph provides insight in the execution structure of block diagram models. A function graph representation for generic blocks in a block diagram formalism is given, as well as a representation for Simulink blocks. It is shown how a block can, in principle, have an arbitrary number of functions capturing its behavior, while in Simulink, in the context of the work presented in this paper, at most two such functions are allowed. It is then shown that in forming a new block by aggregation, the aggregation of functions along the f/g diametry may result in circular dependencies. An alternate aggregation is demonstrated to eliminate these dependencies and to result in an aggregate block that is more robust in terms of execution dependencies.

REFERENCES

Lee, E. A. 2000, September. What's ahead for embedded software. *Computer* 33 (9): 18–26.

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

Liu, J., and E. A. Lee. 2000, September. Component-Based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics. In *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*, 95–100. Anchorage, Alaska.

Mosterman, P. J., and J. E. Ciolfi. 2004, December. Interleaved Execution to Resolve Cyclic Dependencies in Time-Based Block Diagrams. In *Proceedings of the 43rd IEEE Conference on Decision and Control*. Bahamas.

Nilsson, H., J. Peterson, and P. Hudak. 2003, January. Functional hybrid modeling. In *Lecture Notes in Computer Science*, Volume 2562, 376–390. New Orleans, LA: Springer-Verlag. Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages.

Oberschelp, O., A. Gambuzza, S. Burmester, and H. Giese. 2004, July. Modular generation and simulation of mechatronic systems. In *Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI)*. Orlando.

Simulink 2004, June. *Using simulink*. Natick, MA: The MathWorks.

Wijbrans, K. 1993. *Twente hierarchical embedded systems implementation by simulation: a structured method for controller realization*. PhD dissertation, University of Twente, Enschede, The Netherlands. ISBN 90-9005933-4.