Using Interleaved Execution to Resolve Cyclic Dependencies in Time-Based Block Diagrams

Pieter J. Mosterman and John E. Ciolfi

***Abstract*— Time-based block diagrams are a convenient formalism for describing dynamic systems such as those found in control system design. The block-diagram model of the control system is used to capture implementation effects such as sample rates and fixed-point data types. One implementation decision concerns the desired software partitioning used in performing simulation and program synthesis. This paper shows how software partitioning may yield undesirable side-effects involving 'false' algebraic loops, which affects model execution and program synthesis. It presents an approach to resolve these pathological cases for a class of block diagram models.**

I. INTRODUCTION

The use of block diagrams to design control systems is prevalent in standard control text-books (e.g., [1], [2]), where they typically are used as the mathematical model of a control system. The block diagram formalism has been increasingly extended (e.g., by adding different subsystem types) to capture the growing complexity of control systems.

At present, custom-off-the-shelf tools (e.g., Simulink® [8]) support the control system design process very well using such time-based block diagrams, extended with means for partitioning and hierarchy.

Since much of the system control is implemented using embedded computing, the behavioral design often has to be transformed into an implementation in software [4], [5]. This is the domain of the software engineers. In a streamlined design process, the control design engineers and the software engineers use the same model.

This process requires implementation aspects to be included in the models. For example, the control law may be implemented on a fixed-point processor and the fixed-point representation may affect controller performance. Other implementation decisions are sample rates, supervisory logic, and data types. These implementation choices often affect the system being modeled. For example, discretization of a continuous control law may lead to additional delays that require re-tuning of the original parameters. The more such detail can be modeled, the smaller the leap of faith between the controller model, its simulation and test results, and the software implementation of the modeled system.

A more fundamental implementation effect has to do with the software partitioning used in executing the block diagram. A time-based block diagram is a dynamic system that consists of blocks that define input-output signal relationships for each point in time and for each signal. The blocks are themselves dynamic systems. A dynamic system is represented by a set of equations. In case a state space description is applied, a block in the continuous-time domain consists of output equations and state derivative equations. A block in the discrete-time domain consists of output equations and state update equations. A hybrid block consists of output, state update, and state derivative equations. Additional equations that define relationships between the signals and states can be imposed; for example, zero crossing equations define a relationship that is used to detect modal changes.

A dynamic system is realized in software by using block methods that capture the block equations, implemented as software functions 'attached' to the blocks. For example, a discrete delay block (z^{-1}) has an output method that copies its state to its output signal and an update method that copies its input to its state to advance its discrete state, which corresponds to the state space description

$$y(t) = x(t) \tag{1}$$

$$x(t+h) = u(t) \tag{2}$$

where t is time, h is the sample time, $y(t)$ is the output signal, $x(t)$ is the state, and $u(t)$ is the input.

Solving a dynamic system involves solving the output equations and then the state equations (update, derivative, etc.) in a loop that advances time. The initial state is given at the start of the simulation. In software, the equations are solved in this loop by invoking the block methods by type in a predefined order that is consistent with systems theory.

This paper presents an adverse effect of decisions about the structure of software used to implement a control system design. It explains how this effect emerges as 'algebraic loops' in the design and shows how it can be eliminated for a class of models. The method is implemented in Simulink.

Section II presents the principles of execution of block diagram models as implemented by Simulink and the 'algebraic loop' concept. Section III shows how implementation choices about the software architecture may introduce algebraic loops. Section IV presents an interleaved execution strategy that resolves such implementation-induced algebraic loops. In Section V this approach is illustrated for hierarchical models. Section VI presents conclusions.

II. PRELIMINARIES

Block diagrams are a *causal* formalism because they are based on transformations with well-defined input and output signals. So, the blocks compute output, y, based on input, u, state, x, and the independent variable time t,

P. J. Mosterman and J. E. Ciolfi are with The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA [pieter.mosterman|john.ciolfi]@mathworks.com

i.e. $y = f(x, u, t)$. This approach contrasts with non-causal formalisms such as SimMechanics [7] where a multi-body topology is captured by connecting joints and bodies without explicitly stating whether one body computes its force or velocity from another. Other non-causal formalisms include bond graphs [6] and Modelica [3]. These formalisms are often best used for plant modeling.

A block diagram, then, consists of blocks with their input and output connected. For example, in Fig. 1 the block A produces a constant value 1 and is connected to a sum block, B, that adds the output of A and the output of the gain block E. The input of E is multiplied by a constant 1 to produce the output. The output of B is connected to the input of the *subsystem* C. Inside this subsystem,[1] the input is represented by the block *In* that connects to the gain *Gain* with gain factor 1. The output of *Gain* is the input of a unit delay block, *Delay*. This block delays its input by one sample, where the sample rate can be user-selected or inherited. Finally, the outport *Out* emits the output of *Delay* from C. It is then fed back through the gain E, as well as made available to the user by the outport block D.

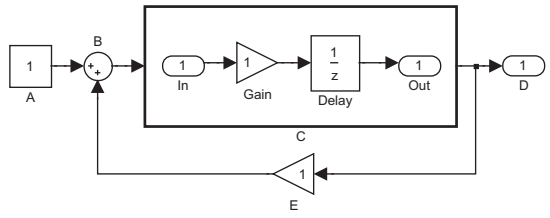


Fig. 1. A block diagram.

Solving the block diagram involves repeatedly solving the block output equations, advancing the state, and incrementing time. The output equations can be solved by iterating over them till there is no change in the output signals. A more efficient execution first orders each type of the block methods, where, ideally, iteration is not required. For example, a discrete model has output and update methods, and the execution loops uses an output block method execution list and an update block method execution list.

The block connections do not necessarily define the order of execution of the individual block equations (methods). A directed graph is derived from the block diagram first to obtain the block *sorted order* that produces the block method execution lists. The directed graph contains vertices that represent the blocks with run-time (output, update, derivative, etc.) methods. The edges are formed from the block methods that have direct feedthrough, i.e., output equations that access the current input, $u(t)$. Edges in the graph are added to capture implicit direct feedthrough constraints imposed by structural elements such as strong grouping within the block diagram (nonvirtual subsystems). When the directed graph is sorted, it may result in *strongly connected components*, i.e., a subset of vertices such that any vertex is reachable from any other vertex in the subset, and the subset is not a subset of a larger set. These strongly connected components are algebraic relationships among the output equations. They are referred to as *algebraic loops*. Because of the cyclic dependency, the values of variables in an algebraic loop cannot be computed by a set of forward computations, and iteration is required.

In Fig. 1, the sorted order is A, *Delay*, D, E, B, *Gain*. The inport, *In*, the outport, *Out*, as well as the subsystem, C, do not show up in this sorted list because they are 'virtual' elements without run-time methods. Therefore, the hierarchical structure that C creates can be flattened when generating the sorted order. The block method execution lists can now be formed by sequencing over the block sorted order by method type. For example, in discrete-time systems such as that shown in Fig. 1, the output block method execution list is A_{op}, Delay_{op}, D_{op}, E_{op}, B_{op}, Gain_{op}, and the update block method execution list is simply Delay_{up}, because this is the only block here with an update method. This paper discusses only block diagrams with discrete-time blocks.

One execution step then first computes $A_{op} \rightarrow \mathit{Delay}_{op} \rightarrow D_{op} \rightarrow E_{op} \rightarrow B_{op} \rightarrow \mathit{Gain}_{op}$. Next, *Delay* is updated by copying the output of *Gain* into its state, Delay_{up}. The automatically generated code for the step function is a direct reflection of this analysis

```
void gaindelay01_step(void)
{

   /* UnitDelay: '<S1>/Delay' */
   gaindelay01_B.Delay = gaindelay01_DWork.Delay_DSTATE;

   /* Outport: '<Root>/D' */
   gaindelay01_Y.D = gaindelay01_B.Delay;

   /* Gain: '<Root>/E' */
   gaindelay01_B.E = gaindelay01_B.Delay * gaindelay01_P.E_Gain;

   /* Sum: '<Root>/B' */
   gaindelay01_B.B = gaindelay01_P.A_Value + gaindelay01_B.E;

   /* Gain: '<S1>/Gain' */
   gaindelay01_B.Gain = gaindelay01_B.B * gaindelay01_P.Gain_Gain;

   /* Update for UnitDelay: '<S1>/Delay' */
   gaindelay01_DWork.Delay_DSTATE = gaindelay01_B.Gain;

   /* (no update code required) */
}
```

The output of the constant block is eliminated, as it is constant during simulation.

In case code is generated for the two separate execution stages, the output code looks like this

```
void gaindelay01_output(void)
{

   /* UnitDelay: '<S1>/Delay' */
   gaindelay01_B.Delay = gaindelay01_DWork.Delay_DSTATE;

   /* Outport: '<Root>/D' */
   gaindelay01_Y.D = gaindelay01_B.Delay;

   /* Gain: '<Root>/E' */
   gaindelay01_B.E = gaindelay01_B.Delay * gaindelay01_P.E_Gain;

   /* Sum: '<Root>/B' */
   gaindelay01_B.B = gaindelay01_P.A_Value + gaindelay01_B.E;

   /* Gain: '<S1>/Gain' */
   gaindelay01_B.Gain = gaindelay01_B.B * gaindelay01_P.Gain_Gain;
}
```

while the update code is

```
void gaindelay01_update(void)
{
```

[1] In Simulink this transparent view is not available, so the figure has been edited for ease of understanding.

```
    /* Update for UnitDelay: '<S1>/Delay' */
    gaindelay01_DWork.Delay_DSTATE = gaindelay01_B.Gain;
}
```

Program synthesis (or automatic code generation) for embedded controllers requires that each execution step have a fixed upper bound in terms of computational complexity so that it can be executed in real time. This means that, computations that require iteration with an unknown upper limit are not suitable, and real-time code for models with *algebraic loops* cannot be automatically synthesized.

If in Fig. 1 the unit delay *Delay* were removed (i.e., *Gain* is directly connected to *Out1*), the circular dependency does not allow a sorted order to compute all signals in one pass. The circular dependency manifests itself because the input to, say, B, requires the output of E, which requires the output of the subsystem, C, and with *Delay* removed, this requires the output of *Gain*. However, to compute the output of *Gain*, the output of B, which was to be computed, must be known to begin with.

III. IMPLEMENTATION-INDUCED ALGEBRAIC LOOPS

To make the step between the modeled design of a controller and the corresponding software as small as possible, it is advantageous to capture the software partitioning via the control model and analyze its feasibility. Catching implementation difficulties early in the design process is less expensive than when they emerge further downstream.

Simulink supports this need by providing *nonvirtual* subsystems that *strongly* group blocks together (e.g. enabled subsystems) and that let supervisory logic control the evaluation of the blocks within the nonvirtual subsystems. Furthermore, Simulink provides nonvirtual function-call subsystems that may be executed by Stateflow® [9] charts based upon user-defined logic. Nonvirtual subsystems are beneficial in system modeling because they can capture significant system changes (e.g., automobile clutch engagement).

Nonvirtual subsystems introduce hierarchy in the execution of the block diagram whereby block method execution lists are associated with each nonvirtual subsystem. In particular, nonvirtual subsystems ensure that the block methods of the constituent blocks are executed within their own context. Thus, in the simulation or generated code, a nonvirtual subsystem that contains blocks with an output method and an update method has these methods itself.

If in Fig. 1 the subsystem C is made a nonvirtual *atomic* subsystem, the output methods of its constituents (*Gain* and *Delay*) should be executed consecutively. The top level of the block diagram has output block method list: $A_{op} \rightarrow B_{op} \rightarrow C_{op} \rightarrow E_{op} \rightarrow B_{op}$ where B, C, and E form an algebraic loop. The atomic subsystem corresponding to C_{op} has output block method list: $Gain_{op} \rightarrow Delay_{op}$ for the internals of C. The output call C_{op} is directed to execute the corresponding block method execution list, $Gain_{op} \rightarrow Delay_{op}$.

Sequential execution of the top-level execution list would produce erroneous results. The sorted order derived in Section II has D, E, and B computing their output in between *Delay* and *Gain*. In the newly derived execution sequence, however, this is impossible because the implementation decision that C should be atomic, making it one function call in the design, confounds the necessity to intersperse the output calls of its constituents with those of other blocks: Making C atomic has created an algebraic loop, i.e., the input of C has to be available when its output is computed. Only when the input to C is available when the output of *Delay* is to be computed, can the output of *Delay* and *Gain* be computed consecutively.

The algebraic loop in this system has proved counter-intuitive because of the presence of the unit delay. Superficial inspection leads to the observation that no *direct feedthrough* from input to output exists for the subsystem C, and, therefore, it cannot be part of an algebraic loop.

More intuitive are subsystems that belong to another class of algebraic loops that emerge when they are made atomic. For example, in Fig. 2 the constant, A, connects to a subsystem B, in which *Gain1* computes an output based on the connected constant. This output is fed back into B by the gain D, and *Gain2* computes the other output of B, to be displayed by the scope C. In case B is not atomic, this corresponds directly to the derived sorted order: $A \rightarrow$ *Gain1* $\rightarrow D \rightarrow$ *Gain2* $\rightarrow C$.

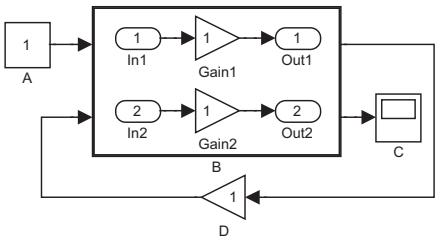


Fig. 2. Another algebraic loop.

If B is made atomic, an algebraic loop emerges and there is no sorted order that allows a straightforward computation of all signals. For this class of models the underlying problem is more obvious, as each input port has direct feedthrough to an output port. An evaluation of pairs of input ports and output ports could reveal that an underlying ordering could apply, yet it would still not be applicable, as the subsystem B has to be executed as one inseparable unit. Exploiting the input-to-output port information requires partial execution of B as well, which violates the implementation requirement that all of B constitute one function in software.

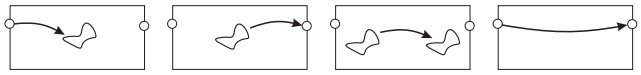
The first class of problems arises because of cyclic dependencies *across the two execution stages*, i.e., the output and update call. The second class of problems arises because of cyclic dependencies *within one execution stage*, i.e., the output call.

Figure 3 presents the classification, showing an ontology

of connection cases:

- In Fig. 3(a) the input is connected to some computations that do not produce output based on these input.
- In Fig. 3(b) there are internal computations that produce an output based on, for example, their states, but not their input.
- In Fig. 3(c) internal computations connect to other internal computations, and thus can be combined into one set of internal computations.
- In Fig. 3(d) there is a direct dependency between output values and input values.

Of these, the first and last class (Fig. 3(a) and Fig. 3(d)) have direct feedthrough, though of different types.



(a) Input sink. (b) Output source. (c) Internal. (d) External.

Fig. 3. Ontology of subsystem connection relations.

IV. ELIMINATING A CLASS OF ALGEBRAIC LOOPS

To enable program synthesis for the class of models that appear not to have direct feedthrough from input to output, shown in Fig. 3(a) and illustrated in Fig. 1, it is observed that internal blocks with no direct feedthrough (there must be at least one), do not need their input in the execution stage that computes output. Only in the update stage is it necessary for those blocks to have the correct input.

A. Principle of the Solution

A solution can 'move' output computations from the output stage of the execution to the update stage. If the output of blocks that relate to internal variables only is computed immediately before the update, these variables have correct values and the algebraic loop has been eliminated.

For example, in Fig. 1 the *Gain* block inside C needs to compute its output before the *Delay* block is updated. However, the output of *Gain* does not need to be computed in the output stage of the execution, but can be done immediately before the *Delay* is updated, i.e., in the update stage of the execution.

To implement this functionality, the *Gain* block could register its output computation as an update call. An execution step then consists of two stages:

1) The output stage is performed based on the sorted list with C an entity because atomic, $A_{op} \rightarrow \overset{C_{op}}{\boxed{Delay_{op}}} \rightarrow D_{op} \rightarrow E_{op} \rightarrow B_{op}$.
2) The update stage is performed based on the sorted list as well, $\overset{C_{up}}{\boxed{Gain_{op} \rightarrow Delay_{up}}}$

Note that the update call of C contains the output call of *Gain*. Stepping through this execution, it is clear that the output stage produces a correct value on the scope, D, and computes a correct value on the input of the subsystem, C. In the update stage, this value is then 'pulled in' by *Gain* before *Delay* is updated, and so, again, the correct value is used.

Because the solution exploits the update stage of the execution, it only applies if no direct feedthrough exists from input to output. The input can be connected to a number of blocks with direct feedthrough but, ultimately, a non-direct feedthrough block as to be present before the signal is output. Therefore, the solution only applies to subsystems with internal connections of the *input sink* type as shown in Fig. 3(a). Note that the term sink is used in a slightly broader sense than as strictly defined by Simulink. In this context, a sink is a block with direct feedthrough. Subsystems with connection of the sort in Fig. 3(d) require a different approach, which is beyond the scope of this paper.

B. Mechanics of the Solution

In general, there may be many direct feedthrough blocks before a non-direct feedthrough block, a 'sink', breaks the loop. The output of all these blocks needs to be evaluated before the update of the sink block is computed. Additionally, multiple input and output ports may be present and, compounding the issue, the classes of connections in Fig. 3 may coexist within a subsystem and even interact with each other.

To find the appropriate set of blocks whose output needs to be computed immediately before the first update of a block in the same subsystem, a depth first search is initiated from each input port. When the search reaches a sink, it recurses back, marking each of the traversed blocks as visited and a potential loop breaker. If, on the other hand, it reaches an output port, each of the traversed blocks is marked visited but not a loop breaker. If a block is reached that has been visited already, its marking is adopted and assigned to the blocks traversed so far.

A two-pass approach is required. The first analysis may mark blocks as potential loop breakers, but they may be connected to an input port that is direct feedthrough along an alternate path. Simply executing the blocks marked 'loop breaker' results in patches of output computations in the update stage that are unnatural.

For example, in the system in Fig. 4, the first pass starts at the input port *In* of the subsystem C. It traverses *Gain* and *Gain1* before the sink *Delay* is reached. It then recurses back up, marking *Gain1* a loop breaker, and then initiates another depth first search along the second branch. This branch connects to an output port. Therefore, when it is recursed back up, *Gain2* and *Gain* are not marked loop breakers, and *In* is marked to have direct feedthrough. However, *Gain1* is still marked a loop breaker.

This situation can be handled in a number of different ways (e.g., adding the blocks to the 'update' list can be

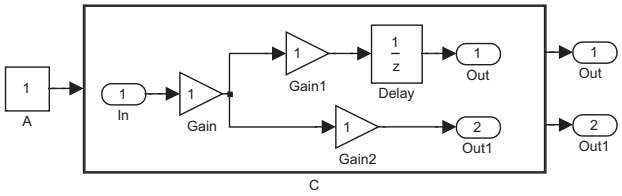


Fig. 4. A two-pass analysis is required.

sequential starting from the input ports, in which case *Gain1* is never found, as *In* is a direct feedthrough port). In the present approach, all visited blocks are marked appropriately, taking into consideration that *In* is set to be direct feedthrough and, therefore, marks all blocks that it traverses not to be loop breakers.

C. Implementation of the Solution

Instead of individually registering an update call for each of the loop-breaking blocks and then rerouting this call to the output call, a more structured approach is implemented that requires only one such rerouted call. It accumulates all loop-breaking blocks in an atomic subsystem and moves these into an automically synthesized subsystem. This subsystem has no graphical representation but is present only in the computational structure.

Figure 5 shows an example with the hidden subsystem visualized. Here, the same system from Fig. 4 is used, but an additional unit delay block, *Delay1* is inserted in the lower branch of C. Now *Gain*, *Gain1*, and *Gain2* are loop breakers, and, therefore, are accumulated into an atomic subsytem called *TmpSynthesizedDirectFeedthroughAtomicSubsystem*.

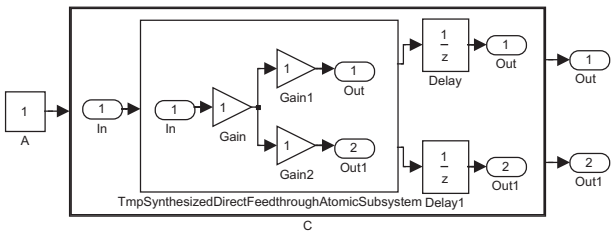


Fig. 5. An atomic subsystem is synthesized.

Now, only the synthesized subsystem needs to register an update call and then rerout it to calling the output of its constituents. For the system in Fig. 5, this results in the following execution structure for the output call

$$A_{op} \rightarrow \overset{C_{op}}{\boxed{Delay_{op} \rightarrow Delay1_{op}}} \rightarrow \mathit{Out} \rightarrow \mathit{Out1} \tag{3}$$

and for the update call

$$\overset{C_{up}}{\boxed{\mathit{TmpSynthesizedDirectFeedthroughAtomicSubsystem}_{up} \; \boxed{\mathit{Gain}_{op} \rightarrow \mathit{Gain1}_{op} \rightarrow \mathit{Gain2}_{op}}}} \rightarrow Delay_{up} \rightarrow Delay1_{up} \tag{4}$$

The automatically generated code for this model reflects the output code that has been moved into the update call. The output call then contains

```
void solved_output(void)
{

   /* Outputs for atomic system: '<Root>/C' */

   /* UnitDelay: '<S1>/Delay' */
   solved_B.Delay = solved_DWork.Delay_DSTATE;

   /* UnitDelay: '<S1>/Delay1' */
   solved_B.Delay1 = solved_DWork.Delay1_DSTATE;

   /* Outport: '<Root>/Out' */
   solved_Y.Out = solved_B.Delay;

   /* Outport: '<Root>/Out1' */
   solved_Y.Out1 = solved_B.Delay1;
}
```

while the update call becomes more involved

```
void solved_update(void)
{

   /* Update for atomic system: '<Root>/C' */

   {

      /* Outputs for atomic system: 'synthesized block' */

      /* Gain: '<S1>/Gain' */
      solved_B.Gain = solved_P.A_Value * solved_P.Gain_Gain;

      /* Gain: '<S1>/Gain1' */
      solved_B.Gain1 = solved_B.Gain * solved_P.Gain1_Gain;

      /* Gain: '<S1>/Gain2' */
      solved_B.Gain2 = solved_B.Gain * solved_P.Gain2_Gain;
   }

   /* Update for UnitDelay: '<S1>/Delay' */
   solved_DWork.Delay_DSTATE = solved_B.Gain1;

   /* Update for UnitDelay: '<S1>/Delay1' */
   solved_DWork.Delay1_DSTATE = solved_B.Gain2;
}
```

In a larger software architecture, this system would not introduce an algebraic loop.

V. Handling Hierarchy

In case of hierarchy, the function call routing has to assume an interleaving structure between the update and output calls of contained subsystems.

In Fig. 6, the feedthrough analysis is performed from the bottom up on each subsystem. The feedthrough flag of subsystems is set correctly when the analysis is performed on its parent.

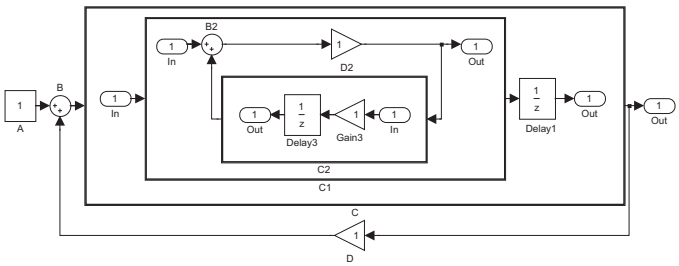


Fig. 6. Nested cyclic dependencies.

For the system in Fig. 6 this results in execution of the output stage according to

$$A_{op} \rightarrow \overset{C_{op}}{\boxed{Delay1_{op}}} \rightarrow \mathit{Out} \rightarrow D_{op} \rightarrow B_{op} \tag{5}$$

but the update stage given in Table I is more complicated, involving a number of interspersed output and update calls. The call sequence starts at the top-left entry, C_{up}, the update call on the subsystem C. This invokes the *output* call of the synthesized subsystem around *C1* to perform the necessary

TABLE I

EXECUTION OF THE UPDATE STAGE.

C_{up}		
$Tmp_{op} \longrightarrow$	$Tmp_{up} \longrightarrow$	$Delay1_{up}$
$C1_{op}$	$C1_{up}$	
$C2_{op} \rightarrow B2_{op} \rightarrow D2_{op}$	$C2_{up}$	
$Delay3_{op}$	$Tmp_{op} \rightarrow Delay3_{up}$	
	$Gain3_{op}$	

output computations. These are triggered by calling $C1_{op}$, which calls $C2_{op}$. Since *Gain3* is part of a synthesized subsystem as well, $C2_{op}$ only needs to execute $Delay3_{op}$. This concludes the $C2_{op}$ call, and next $B2_{op}$ and $D2_{op}$ are executed to complete $C1_{op}$.

At this point, the output call of the synthesized subsystem has been completed, and its update call is made. This call runs down the hierarchy of subsystems. The update call of the synthesized subsystem around *Gain3* is rerouted to its output, and, therefore, calls $Gain3_{op}$. Once $Gain3_{op}$ is computed, the $Delay3_{up}$ call can be made. Similarly, once $C1_{up}$ is completed, $Delay1_{up}$ can be computed.

The automatically generated code for the system in Fig. 6 reflects these computations faithfully. The output stage contains

```
void nesting_output(void)
{

   /* Outputs for atomic system: '<Root>/C' */

   /* UnitDelay: '<S1>/Delay1' */
   nesting_B.Delay1 = nesting_DWork.Delay1_DSTATE;

   /* Gain: '<Root>/D' */
   nesting_B.D = nesting_B.Delay1 * nesting_P.D_Gain;

   /* Sum: '<Root>/B' */
   nesting_B.B = nesting_P.A_Value + nesting_B.D;
}
```

The more elaborate update stage is

```
void nesting_update(void)
{

   /* Update for atomic system: '<Root>/C' */

   {

      /* Outputs for atomic system: 'synthesized block' */

      /* SubSystem: '<S1>/C1' */

      /* Outputs for atomic system: '<S2>/C2' */

      /* UnitDelay: '<S3>/Delay3' */

      nesting_B.Delay3 = nesting_DWork.Delay3_DSTATE;

      /* Sum: '<S2>/B2' */

      nesting_B.B2 = nesting_B.B + nesting_B.Delay3;

      /* Gain: '<S2>/D2'
      nesting_B.D2 = nesting_B.B2 * nesting_P.D2_Gain;

      /* end of Outputs for SubSystem: '<S1>/C1' */
   }

   /* atomic SubSystem Block: <S1>/TmpSynthesizedDirectFeedthroughAtomicSubsystem */

   /* Update for atomic system: 'synthesized block' */

   /* Update for SubSystem: '<S1>/C1' */

   /* Update for atomic system: '<S2>/C2' */

   {

      /* Outputs for atomic system: 'synthesized block' */

      /* Gain: '<S3>/Gain3'
      nesting_B.Gain3 = nesting_B.D2 * nesting_P.Gain3_Gain;
   }

   /* Update for UnitDelay: <S3>/Delay3 */
   nesting_DWork.Delay3_DSTATE = nesting_B.Gain3;

   /* end of Update for SubSystem: '<S1>/C1' */

   /* Update for UnitDelay: <S1>/Delay1 */
   nesting_DWork.Delay1_DSTATE = nesting_B.D2;
}
```

VI. CONCLUSIONS

An important stage in the design of embedded control systems is taking the model of the designed controller to a software implementation. However, imposing implementation constraints on the designed controller may result in complications, for example, because fixed-point data types are used or because the sorted order is affected. It is beneficial to analyze and resolve these complications at a model level, to prevent the more costly iteration between generating code, establishing the problem, relaying it back to the control design engineers to modify the model for the next pass.

This paper addressed the complication that arises in deriving an explicit sorted order when implementation constraints are included in the model. In particular, an algebraic loop may arise because of these constraints. Rather than forcing a change in the design, such as including unit delay blocks to break the loop or repartitioning the software architecture into smaller pieces, an interleaved execution is developed that can be automatically generated from the block diagram.

The method has been implemented in Simulink, an industrial-strength tool for block diagram modeling that supports analyses, simulation, and automatic code generation. The resulting automatically generated code reflects the desired interleaved execution of output and update calls throughout the execution hierarchy.

REFERENCES

[1] Karl J. Åström and Björn Wittenmark. *Computer Controlled Systems: Theory and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[2] Chi-Tsong Chen. *Linear Systems Theory and Design*. Holt, Rinehart and Winston, Inc., New York, 1970. ISBN 0-03-060289-0.

[3] Hilding Elmqvist *et al*. Modelica^{tm}–a unified object-oriented language for physical systems modeling: Language specification, December 1999. version 1.3, http://www.modelica.org/.

[4] 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.

[5] 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.

[6] Henry M. Paynter. *Analysis and Design of Engineering Systems*. The M.I.T. Press, Cambridge, Massachusetts, 1961.

[7] SimMechanics. *SimMechanics User's Guide*. The MathWorks, Natick, MA, 2002.

[8] Simulink. *Using Simulink*. The MathWorks, Natick, MA, January 2002.

[9] Stateflow. *Stateflow User's Guide*. The MathWorks, Natick, MA, 2002.