

Optimal Scheduling in Graphical Modeling Environments

Michael Burke
The MathWorks

Copyright © 2009 The MathWorks. Published by SAE International with permission.

ABSTRACT

Methods for controlling execution order in traditional text-based languages such as C and Fortran are well established. The transition to graphical programs has revealed some of the hidden issues inherent in any scheduling routine, specifically data dependency and data protection (in multirate systems). Graphical programming languages provided built-in diagnostics that allow users to analyze the data dependencies to develop optimal schedules from a data propagation perspective. This paper examines one heuristic that can be used to develop an optimal schedule for an arbitrary model.

INTRODUCTION

The use of computer-based graphical modeling languages to model control systems has been steadily increasing since the early 1980s. Use of graphical modeling languages is growing, in part, because the graphical representation closely models traditional control design methodologies, making it a favorite tool among control engineers. With the introduction of code generation technologies in the late 1990s, the use of these languages expanded beyond control engineering into the software engineering domain.

Entry into the software engineering domain highlighted the importance of controlling the execution order of the graphical components. Unlike in traditional programming languages such as C or Fortran, where the execution of components (e.g., functions) is explicitly set by a scheduling function, graphical components depend on

the connection between components. The graphical language attempts to determine the correct execution order (sorted order) based on the connectivity of the system (see Figure 1).

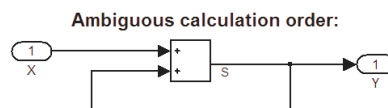


Figure 1. Ambiguous execution order example.

This paper uses Simulink as the reference graphical modeling language. Simulink takes a directed graph approach to determining the execution order of a system. When a system is assembled in such a way that the execution order is ambiguous, the tool infrastructure provides diagnostic error messages (see Figure 2).

Loop_E1			
View Font Size			
	Message	Source	Reported by
	Block error	C	Simulink
	Block error	B	Simulink
	Block error	D	Simulink
	Block error	A	Simulink
	Block error	C	Simulink

Figure 2. Example error message.

The error messages can be interpreted into the node tree of a data dependency graph. Also, the proper location for resolving the loop can be determined by

observing where the flow direction of the graph changes (from input to output or from output to input).

OBJECTIVE - This paper shows how to resolve execution order ambiguity in graphical models by using a simple heuristic based on directed graph methodologies. The resulting model is optimized from the scheduling perspective because it will use the minimum number of loop-breaking blocks (Unit Delay blocks) to resolve the loops. In addition, by following the heuristic method, the end user will have a better understanding of the interdependency of his or her system.

OVERVIEW OF DATA DEPENDENCIES

Data dependency is the requirement that for any calculation, all the values on the right-hand side (RHS) of an equation are known prior to starting the calculation. The value on the left-hand (LHS) side is dependent on the values on the right hand side. In a subsystem context, the concept of calculation order is equivalent to execution order of the subsystems.

$$LHS_n = f(RHS_n)$$

The C language does not prevent users from writing equations in which the LHS is assigned before the RHS. This means that old or noninitialized data can be used, which can have unexpected or incorrect results.

$$LHS_n = f(RHS_{n-1})$$

$$LHS_n = f(??)$$

Graphical programming languages use data dependency to determine the order in which calculations are performed. If Block B uses the output of Block A, then Block B is said to be dependent on Block A. Dependency propagates through blocks, so if C uses B's output then C depends on A (see Figure 3).

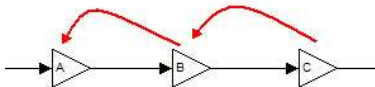


Figure 3. Data dependencies.

$$\begin{aligned} B(n) &= f(A(n)) \\ C(n) &= f(B(n)) \\ C(n) &= f(f(A(n))) \end{aligned}$$

Blocks that maintain state information, such as integrators and unit delays, break data dependency for the blocks following them (see Figure 4).

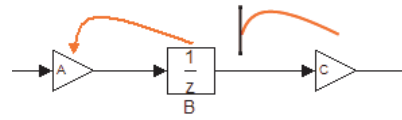


Figure 4. Effect of unit delays.

$$\begin{aligned} B(n) &= f(A(n)) \\ C(n) &= f(B(n-1)) \end{aligned}$$

In Simulink the same principle can be extended to atomic subsystems.

UNDERSTANDING DATA DEPENDENCIES IN GRAPHICAL SYSTEMS - Complications arise when feedback loops are introduced into the system. Again, using a simple block example, we can show how feedback loops introduce ambiguous execution order into the system (see Figure 5).

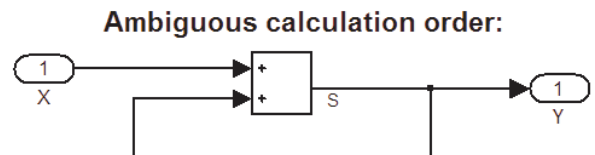


Figure 5. Ambiguous calculation order.

Note: The subscript letter "i" indicates the iteration count.

$$\begin{aligned} Y_i &= S_i \\ S_i &= X_i + S_i \end{aligned}$$

Subtracting S_i from both sides (see Figure 6) results in:

$$X_i = 0$$

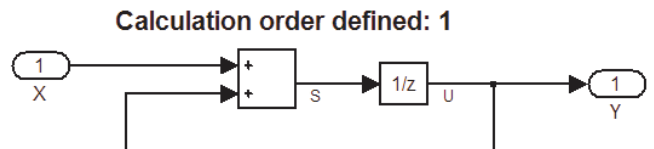


Figure 6. Data dependencies resolved, example 1.

$$\begin{aligned} Y_i &= U_i \\ U_i &= S_{i-1} \\ S_i &= X_i + U_i \end{aligned}$$

By substitution:

$$Y_i = X_{i-1} + Y_{i-1}$$