

Techniques for Generating and Measuring Production Code Constructs from Controller Models

Bill Chou and Saurabh Mahapatra

The MathWorks

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

ABSTRACT

A key step in Model-Based Design is the deployment of an algorithm as machine code onto a target processor in the production vehicle. Modern software tools automatically generate the algorithmic source code from models. Given the many combinatorial possibilities for realizing a given algorithm within the modeling environment, the generated C source code will be a function of a realization. This dependency is an important consideration because the quality and clarity of the source code impacts the amount of verification and analysis that must be done for production software development. Other factors involved in generating the machine code from the source code, such as compiler optimization and microprocessor architecture, also contribute to this optimization. Organizations that proactively data mine and gather these optimizations into a set of best practices stand to benefit from reduced development times and lower costs. This paper introduces techniques that can be used to generate and measure code constructs used to create a set of best practices for the Simulink modeling environment. The quality of the object code is measured by examining the algorithm compiled within an Integrated Development Environment.

MODEL-BASED DESIGN

Model-Based Design for embedded control systems development involves a process centered on a model—from requirements capture to implementation and test. This model forms the “executable specification” that is used to communicate the desired system performance. The control design is elaborated and continuously tested

against requirements through simulation. Code is generated from models and rapid-prototyping is carried out to assess the performance of the algorithm in a real-time environment. Software-in-the-loop (SIL), processor-in-the-loop (PIL), and hardware-in-the-loop (HIL) testing and verification of the algorithmic code may be done before deployment on the production vehicle.

The use of automatic code generation maintains the link between the model and the generated C source code^[1]. To change the algorithm later in the design process, it is easier to update the model and regenerate the C source code. This method allows the engineer to focus more on integrating algorithmic code and setting up the infrastructure for embedded system deployment^[2].

Figure 1 shows the code generation workflow in Model-Based Design. A and B denote opportunities for optimizing code.

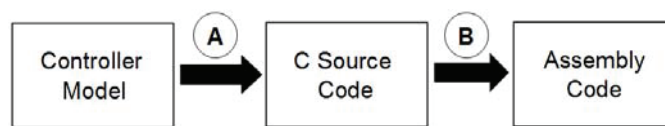


Figure 1. Code generation workflow in Model-Based Design.

For each opportunity, several techniques are available:

- A. Generating C source code from software models:
 - Using modeling design patterns in the controller model
 - Using target-optimized code
- B. Compiling C source code into object code:

- Choosing a microprocessor architecture
- Choosing a compiler
- Using compiler-specific optimizations

Two important metrics to measure the quality of control algorithms running on microprocessors are object code size and execution time.

Object code size is used to measure the quality of the control algorithms, although the proposed techniques can save time as well^[3]. This is due to difficulties in profiling object code. One can look at either the execution time or speed. If time is being measured over several trials, the variability requires looking at the minimum, maximum, or average execution times. If efficiency is being measured by throughput, it is measured differently from execution time. Hence, we use object code size to measure the quality of code.

The following examples illustrate the application of these techniques to optimize object code. Real-Time Workshop Embedded Coder is used to automatically generate C source code from Simulink models. The code is compiled and loaded onto processors supported by Green Hills MULTI. Standard code generation optimization settings, such as expression folding and block reduction, and compiler flags, such as `-a` and `-Osize`, were used unless stated otherwise.

GENERATING C SOURCE CODE FROM SOFTWARE MODELS

Two techniques are available for optimizing the C source code generated from the software model: modeling patterns and target-optimized code.

USING MODELING DESIGN PATTERNS IN THE CONTROLLER MODEL - A modeling design pattern is much like a software design pattern used in object-oriented literature^[4]. It is a template containing modeling elements that can be reused in commonly recurring design problems. Figure 2 shows an example of a Stateflow modeling design pattern for the familiar `do-while` logic. This pattern can be used to generate the common `do-while` loop construct in the C code.

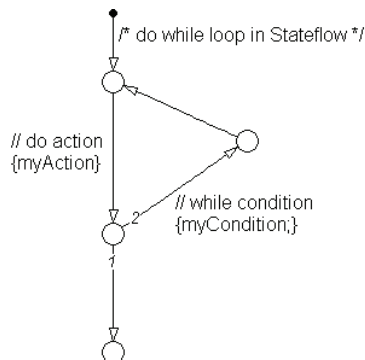


Figure 2. Stateflow `do-while` loop design pattern.

We are interested in modeling design patterns that optimize C source code measured by lines of code (LOC). At a high level, it may lead to more optimized object code.

Figure 3 shows the matrix multiplication of two 10x10 matrices `u_1` and `u_2` in Stateflow. The outer two loops use counters `i` and `j` to loop through rows of `u_1` and columns of `u_2`. The inner-most loop computes each element of the output matrix `y_1` as the dot product of the row from `u_1` and the column from `u_2`. The model uses nested loops very similar to the Stateflow `do-while` loop design pattern shown in Figure 2. The difference lies in the duplicate initializations of `y_1[i][j]` in the outer `i` and `j` loops.

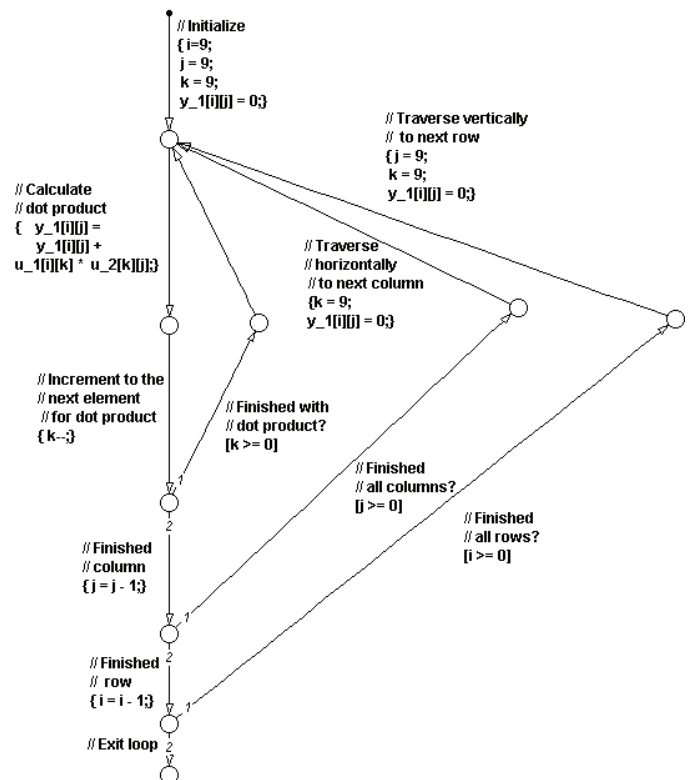


Figure 3. Modeling multiplication of two 10x10 matrices without use of a modeling design pattern.

Figure 4 shows 42 LOC generated from this model. Note the checks for `i` and `j` with redundant initializations of `y_1[i][j]` on lines 32–37 and 41–47. These multiple initializations can be reduced to just one initialization before the `do-while` loop in lines 25–29.

```

9 void DO_step(void)
10 {
11 {
12   int8_T sf_j;
13   int8_T sf_i;
14   int8_T sf_k;
15   int32_T sf_exitg;
16   int32_T sf_exitg_0;
17   sf_i = 9;
18   sf_j = 9;
19   sf_k = 9;
20   y_1[99] = 0;
21   do {
22     sf_exitg = 0;
23     do {
24       sf_exitg_0 = 0;
25       do {
26         y_1[sf_i + 10 * sf_j] = u_1[10 * sf_k + sf_i] * u_2[10 * sf_j + sf_k]
27         + y_1[10 * sf_j + sf_i];
28         sf_k--;
29       } while (sf_k >= 0);
30
31       sf_j--;
32       if (sf_j >= 0) {
33         sf_k = 9;
34         y_1[sf_i + 10 * sf_j] = 0;
35       } else {
36         sf_exitg_0 = 1;
37       }
38     } while ((uint32_T)sf_exitg_0 == 0U);
39
40     sf_i--;
41     if (sf_i >= 0) {
42       sf_j = 9;
43       sf_k = 9;
44       y_1[sf_i + 90] = 0;
45     } else {
46       sf_exitg = 1;
47     }
48   } while ((uint32_T)sf_exitg == 0U);
49 }
50 }

```

Figure 4. Generated C source code without the use of modeling design patterns.

Figure 5 shows an implementation of the same algorithm that makes proper use of a nested Stateflow do-while loop design pattern. The generated C source code (see Figure 6) has only 25 LOC compared with the 42 LOC shown in Figure 4. The redundant initializations of $y_1[i][j]$ and checks for i and j have been eliminated, resulting in more efficient C source code.

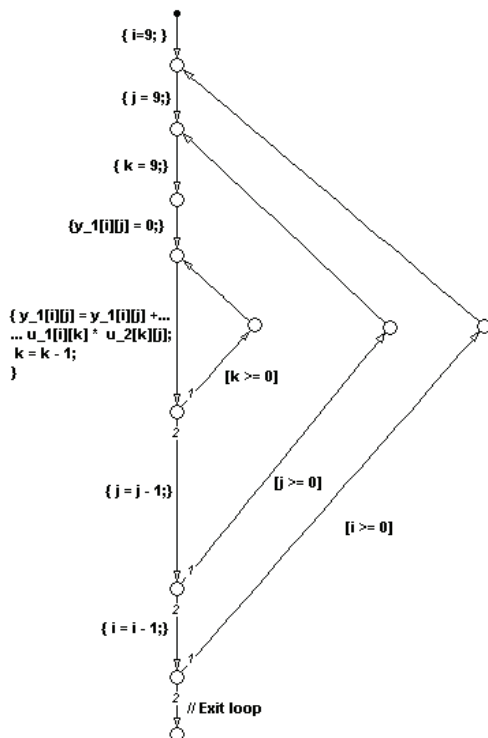


Figure 5. Modeling a multiplication of two 10x10 matrices using nested Stateflow do-while loop design patterns.

```

9 void DO_step(void)
10 {
11 {
12   int8_T sf_j;
13   int8_T sf_i;
14   int8_T sf_k;
15   sf_i = 9;
16   do {
17     sf_j = 9;
18     do {
19       sf_k = 9;
20       y_1[sf_i + 10 * sf_j] = 0;
21       do {
22         y_1[sf_i + 10 * sf_j] = u_1[10 * sf_k + sf_i] * u_2[10 * sf_j + sf_k]
23         + y_1[10 * sf_j + sf_i];
24         sf_k--;
25       } while (sf_k >= 0);
26
27       sf_j--;
28     } while (sf_j >= 0);
29
30     sf_i--;
31   } while (sf_i >= 0);
32 }
33 }

```

Figure 6. Generated C source code with the proper use of nested Stateflow do-while loop design patterns.

The reduced source code contains production code constructs, or C source code constructs in this case, that represent the algorithm more concisely. A subset of mappings from modeling design patterns to common C source code constructs can contain the following list (for other source code languages such as the C++ language, the list may contain different constructs):

- Data types, operators, and expressions such as data declarations, data type conversions, and type qualifiers
- Control flows such as if-then-else, switch, and for-loops
- Functions and program structures such as void-void functions and calling external functions
- Structures such as nested structures and bit fields
- Arrays and pointers

A set for the Simulink modeling environment is available from The MathWorks^[5].

USING TARGET-OPTIMIZED CODE - During the automatic code generation process, it is efficient to replace appropriate sections of the C source code with optimized C code for a specific target. There are two techniques for doing this:

- Reuse existing handwritten or legacy code that has been tested and optimized for a specific target
- Use target-specific libraries that contain mappings of functions and operators to optimized object code

Figure 7 shows ANSI C and optimized C source code for the Infineon TriCore processor.