



# Techniques for Generating Production Code Constructs from Modeling Design Patterns

By Bill Chou

A key step in Model-Based Design is the generation of C code and deployment of an algorithm onto a target processor in the production environment. There are many different ways to implement the same algorithm in a Simulink® model; the quality of the generated C code will depend on the way it was modeled in Simulink.

### Products Used

- Simulink®
- Stateflow®
- Real-Time Workshop Embedded Coder™

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

Using a `do-while` loop as an example, this article focuses on optimizing the algorithm at stage A. We demonstrate how modeling design patterns can be used to generate efficient C code. You can further optimize your algorithms when compiling C source code into object code by using Target Function Libraries. For details, see the paper “Generating Target-Optimized Code Using Target Function Libraries.”

### Modeling Design Patterns and Their Benefits

A modeling design pattern is much like a software design pattern used in object-oriented literature. It is a template containing modeling elements that can be reused in commonly recurring design problems. Figure 2 shows an example of a modeling design pattern for `do-while` logic implemented in Stateflow®. This pattern can be used to generate the common `do-while` loop construct in the C code.

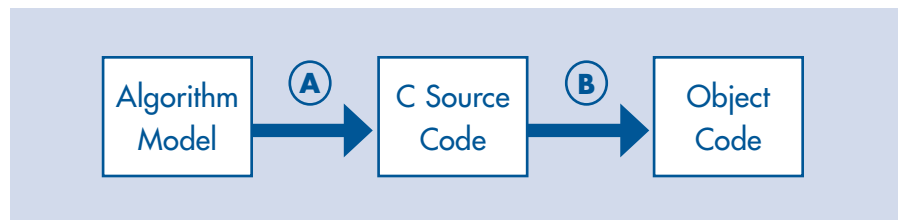


Figure 1. Code generation workflow in Model-Based Design.

Design patterns bring several benefits. First, more optimized C source code can be generated, resulting in designs that utilize less RAM and ROM on the target hardware. Second, modeling patterns can be standardized across models created by various users and teams. This standardization can result in more efficient design reviews because everyone involved is working from familiar modeling patterns. Third, custom modeling patterns can be shared across teams, resulting in the reuse of efficient designs across the organization.

For the design pattern presented in this article, we measure code efficiency in terms

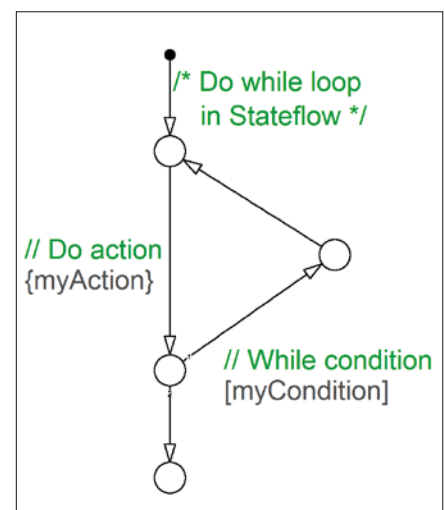


Figure 2. Stateflow `do-while` loop design pattern.

of lines of code (LOC). As a rule of thumb, LOC is a good indication that the source code will produce more optimized object code, although other measures, such as RAM, ROM, stack, and execution time, will ultimately be needed to assess the efficiency of the algorithm running on the hardware.

### Comparing C Code Generated With and Without Design Patterns

Figure 3 shows the matrix multiplication of two 10x10 matrices `u_1` and `u_2` in `Stateflow`. This algorithm could have been implemented using the Matrix Multiplication block, the Embedded MATLAB Function block, or `for`-loops in `Stateflow`. We will use a `Stateflow` chart because this is the clearest way to show the design pattern graphically and to compare the size of the two generated code samples.

Our first example does not make proper use of a design pattern. The two outer loops use counters `i` and `j` to loop through rows of `u_1` and columns of `u_2`. The innermost 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.

This model generates 40 LOC (Figure 4). 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 right before the `do-while` loop in lines 25–29.

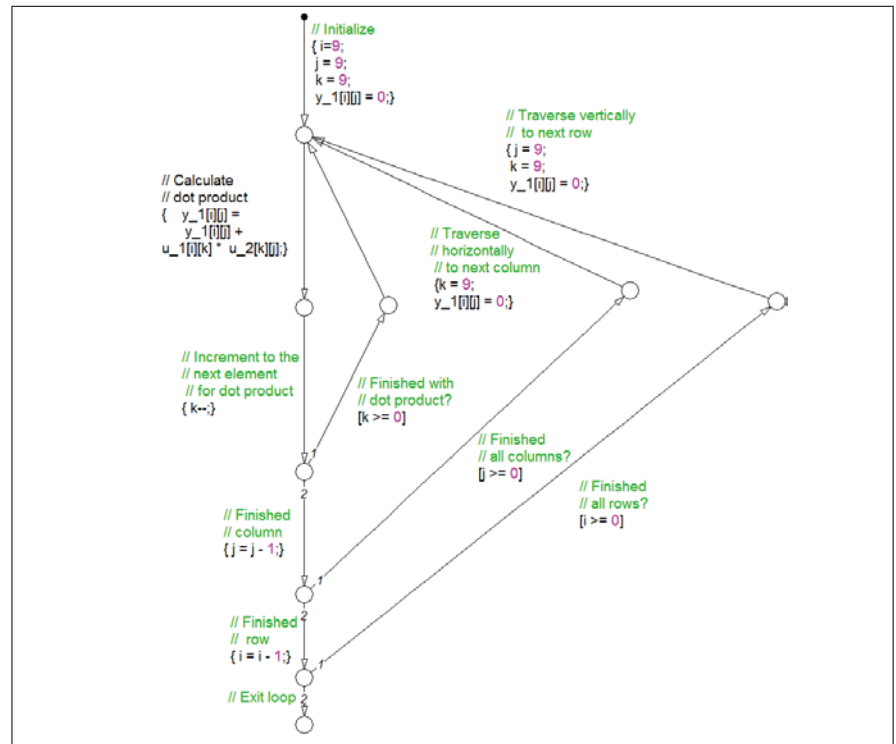


Figure 3. Modeling multiplication of two 10x10 matrices without using a modeling design pattern.

```

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

```

Figure 4. Code generated from an improper design pattern.

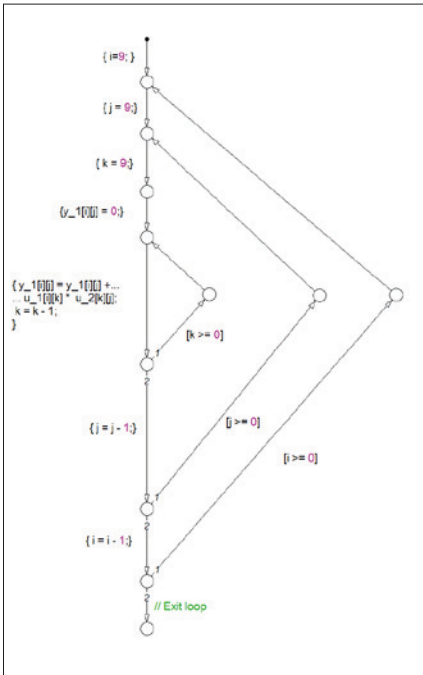


Figure 5. Modeling a multiplication of two 10x10 matrices using nested Stateflow do-while loop design patterns.

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 has only 23 LOC (Figure 6). The redundant initializations of `y_1[i][j]` and checks for `i` and `j` have been eliminated, resulting in more efficient C source code.

The above example shows how modeling design patterns help optimize C source code for size. However, optimized C source code does not necessarily guarantee optimized object code. Other factors, such as the compiling and linking stages, will also affect the quality of the object code. For more information on this topic, refer to “Compiling C Source Code Into Object Code.”

```

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

```

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

### Selecting and Applying Model Design Patterns

You can access additional design patterns using the Stateflow Pattern Wizard or the guide “Modeling Patterns for C Constructs.”

The Stateflow Pattern Wizard, released in R2008b, provides several prespecified design patterns that generate C code constructs such as for-loops, while-loops, and do-while loops (figure 7). Other pat-

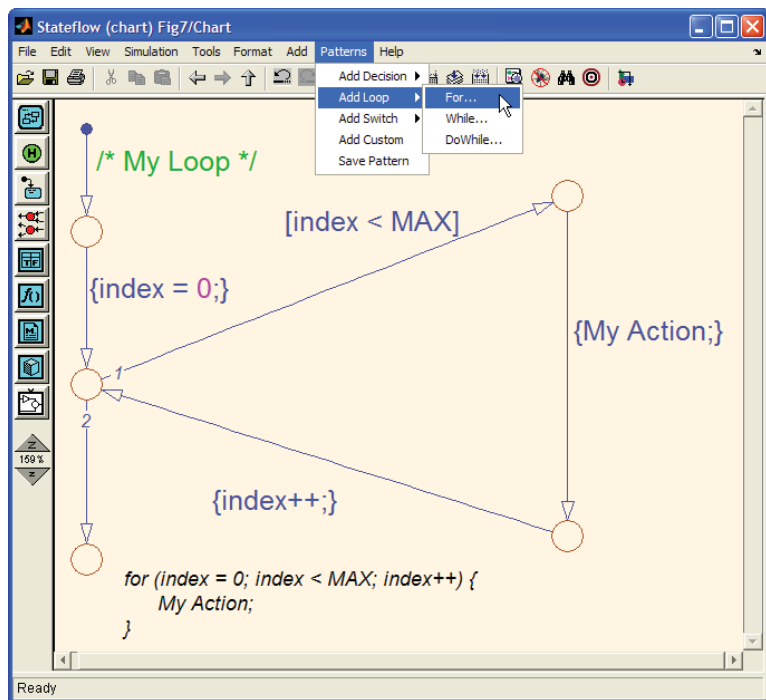


Figure 7. A pre-configured for-while loop design pattern added to a Stateflow chart.

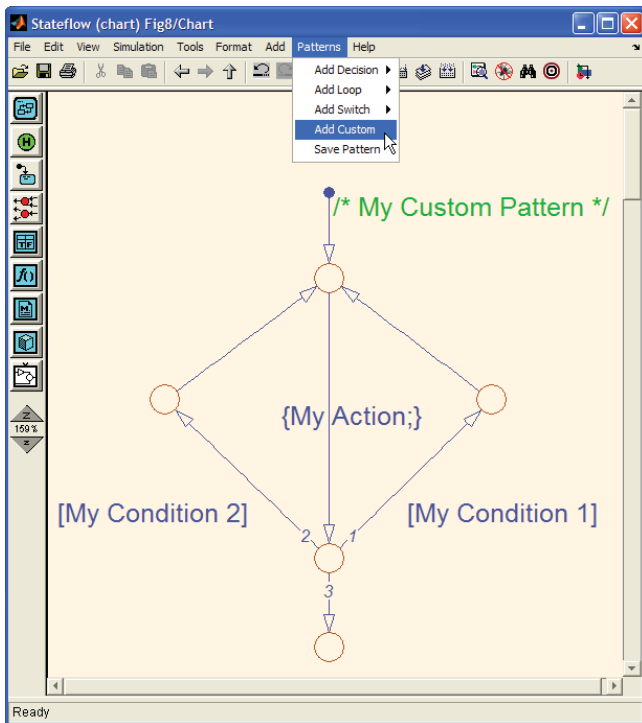


Figure 8. Custom design pattern added to the Stateflow Design Pattern Wizard.

terns, such as those from the “Modeling Patterns for C Constructs” guide, can be programmed into the tool. You can also incorporate your own modeling patterns (Figure 8) and distribute them to others.

“Modeling Patterns for C Constructs” contains mappings from model design patterns to C code constructs for Simulink, Stateflow, and Embedded MATLAB™ functions, such as the `do-while` loop. The following C code construct mappings are included:

- 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

## Getting the Most out of Design Patterns

To gain maximum benefit from design patterns, it is important to understand the various parameters that optimize object code at key steps in the workflow. It is also helpful to incorporate these optimizations into a set of best practices that can be applied to future designs. Documents such as “Modeling Patterns for C Constructs” and “MAAB Modeling Style Guidelines” provide a starting point for building a library of modeling patterns that others can use to generate more efficient C code. These patterns can be distributed through these documents or through tools such as the Stateflow Pattern Wizard. ■

## Resources

### VISIT

[www.mathworks.com](http://www.mathworks.com)

### TECHNICAL SUPPORT

[www.mathworks.com/support](http://www.mathworks.com/support)

### ONLINE USER COMMUNITY

[www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)

### DEMOS

[www.mathworks.com/demos](http://www.mathworks.com/demos)

### TRAINING SERVICES

[www.mathworks.com/training](http://www.mathworks.com/training)

### THIRD-PARTY PRODUCTS AND SERVICES

[www.mathworks.com/connections](http://www.mathworks.com/connections)

### Worldwide CONTACTS

[www.mathworks.com/contact](http://www.mathworks.com/contact)

### E-MAIL

[info@mathworks.com](mailto:info@mathworks.com)

© 2009 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

91767v00 08/09

## For More Information

- **Generating Target-Optimized Code Using Target Function Libraries**  
<http://www.mathworks.com/libraries>
- **Compiling C Source Code Into Object Code**  
<http://www.mathworks.com/compilingc>
- **MathWorks Automotive Advisory Board (MAAB) Standards**  
<http://www.mathworks.com/maabguidelines>