American Astronautical Society
Rocky Mountain Section

AAS 13-031

UNDERSTANDING MODEL AND CODE BEHAVIOR FOR STATEFLOW CONSTRUCTS

William Campbell, Mike Anthony, and Becky Petteys

The MathWorks, Inc.

36th ANNUAL AAS GUIDANCE AND CONTROL CONFERENCE

February 1 - February 6, 2013
Breckenridge, Colorado

Sponsored by
Rocky Mountain Section

AAS Publications Office, P.O. Box 28130 - San Diego, California 92198

UNDERSTANDING MODEL AND CODE BEHAVIOR FOR STATEFLOW CONSTRUCTS

William B. Campbell[*], Mike Anthony,[†] and Becky Petteys[‡]

Scheduling, supervisory logic, and fault management are often the most challenging components of a software design to develop, test, and verify. In a Model-Based Design process that leverages Simulink®, Stateflow® is regularly employed to mitigate these challenges. Its environment provides an infrastructure for developing state machines, truth tables, and flow charts. While such schematics are helpful in understanding complex logical systems, adopting a new modeling schema brings about its own difficulties. A variety of design patterns are available within Stateflow, but what is the precise behavior of a particular pattern, and which is the most desirable under a particular circumstance?

Common Stateflow design constructs are examined within this report. Fundamental architectural decisions such as state actions versus transition actions, events versus transition conditions, and MATLAB® versus C as the action language are explored by examining the performance of each construct. Behavior is studied within the Simulink model as well as the C code derived from Stateflow via Embedded Coder®. Each construct is vetted for consistency with existing Stateflow modeling standards such as the MathWorks Automotive Advisory Board Model Style Guide and the NASA Orion GN&C MATLAB and Simulink Standards. Results demonstrate that there is rarely an unequivocally superior design construct. However, architectures can be optimized based on specific software application, desired system behavior, and the developers' technical background.

INTRODUCTION

Regardless of the particular industry or application, engineering projects are currently experiencing an increasing prevalence of embedded software components. As demands on software developers expand and the complexity of the systems they are responsible for continue to rise, organizations are choosing to adopt a model-based software development process and all the benefits that it brings. Graphical development environments with built-in dynamic simulation capabilities are a cornerstone of Model-Based Design. Depending on the tool employed, the graphical contents of the model convey a particular concept. In the case of Stateflow, diagram constructs represent finite-state machines and flow charts. Semantics within Stateflow denote the rules and actions that govern diagram behavior.

As with any design environment, Stateflow offers the user some degree of freedom as to how an algorithm is to be implemented. The developer wants to understand which option best satisfies their project's needs, but the repercussions of one design pattern versus another might not be readily apparent. Therefore, it is important that development standards are established early in the planning process so as to ensure consistency amongst all team members. This paper examines the impact of

[*] Senior Application Engineer, The MathWorks, Inc., 3 Apple Hill Dr, Natick, MA 01760
Email: will.campbell@mathworks.com. Web Site: www.mathworks.com
[†] Senior Application Engineer, The MathWorks, Inc., 3 Apple Hill Dr, Natick, MA 01760
Email: mike.anthony@mathworks.com. Web Site: www.mathworks.com
[‡] Application Engineering Manager, The MathWorks, Inc., 3 Apple Hill Dr, Natick, MA 01760
Email: becky.petteys@mathworks.com. Web Site: www.mathworks.com

various design patterns for logic-intensive algorithms developed in Stateflow R2012b. Previous publications have discussed design considerations within Simulink[1] [2]; this paper is intended to be complementary. Stateflow patterns will be evaluated based on behavior of the Stateflow diagram, structure of Stateflow-derived C code generated by Embedded Coder, and consistency with existing industry standards (whenever possible).

While it is not possible to exhaustively explore all design choices within Stateflow, the most critical and commonly encountered questions will be addressed. Since project goals vary widely, it is also not possible to definitively advocate for one position over another under all circumstances. Pertinent data will be presented, and the reader will make a final determination as to their preferred design pattern.

SIMULATION AND TESTING SPECIFICATIONS

Unless otherwise noted, all models were constructed in MATLAB R2012b using Simulink 8.0 and Stateflow 8.0 with C syntax. Models were configured to run using a fixed step, discrete solver with a time step of 1 second. Boolean pulse signals that oscillate between zero and one every two seconds were used as inputs to Stateflow diagrams. C code was generated via Embedded Coder with the inline parameters option active. Some code excerpts had the generated commentary removed for brevity.

CHOOSING HOW TO DEFINE TRANSITIONS

When a Stateflow diagram has a state machine comprised of exclusive states, two constructs are available to specify the transition from one state to another. Figures 1 and 2 illustrate these techniques. In Figure 1, the conditions for transitions to and from State1 to State2 are satisfied when the transition condition A is true. In Figure 2, the conditions for transitions are satisfied when the event B occurs. Transition conditions are defined within a set of square brackets whereas events do not require any semantic encapsulation.

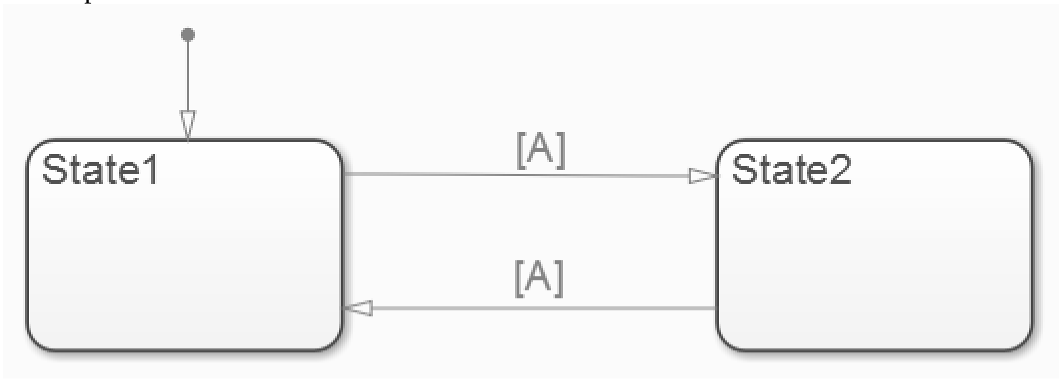


Figure 1. Transition Condition Archetype

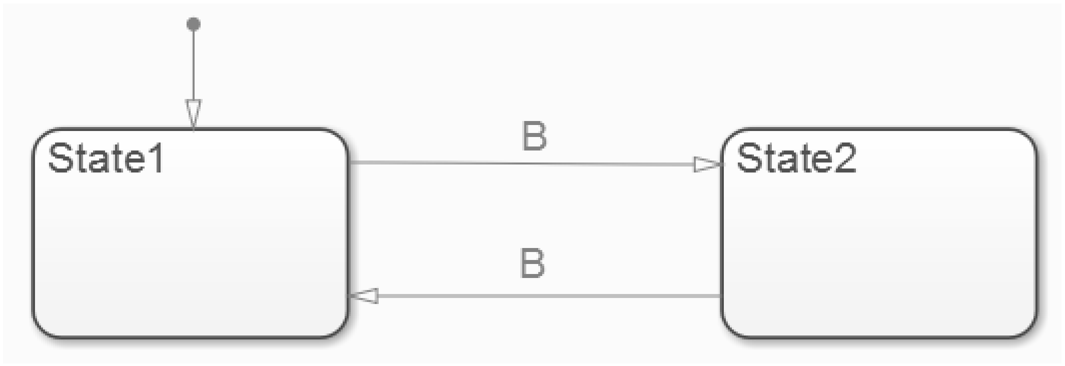


Figure 2. Event-driven Transition Archetype

Transition conditions are logical expressions. They could be more complicated expressions than our single data variable B (i.e. `[(x>=32 && y ~= 8) || myfunc(u) == z)]`). Regardless, the transition condition ultimately evaluates to a Boolean (true or false) value. The key is that this value can be true over extended periods of time, and a transition takes place at any time step Stateflow executes and finds that to be the case. If the Stateflow data variable A is defined to be an input to the diagram and construct a test harness as shown in Figure 3, running the simulation yields the results seen in Figure 4. Stateflow transitions into State2 two seconds into simulation and remains there*, precisely when the value of A is true. At the next simulation time step, three seconds, the diagram departs State2 and return to State1 since A remains true.

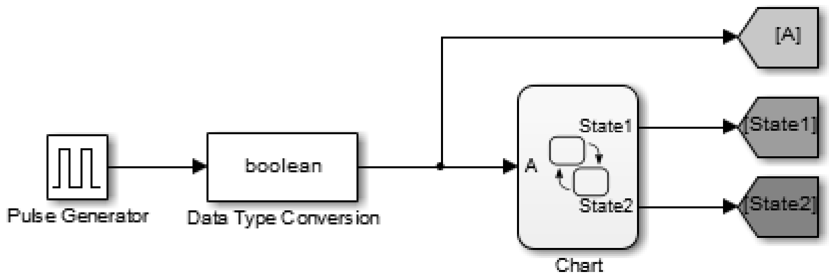


Figure 3. Transition Condition Test Harness

* By default, at least one time step must be spent in a state after it is entered. Thus, even though the transition to State1 is valid two seconds into simulation, the system will not yet depart State2. The user reserves the option to change this behavior and enable "super step semantics." [3]

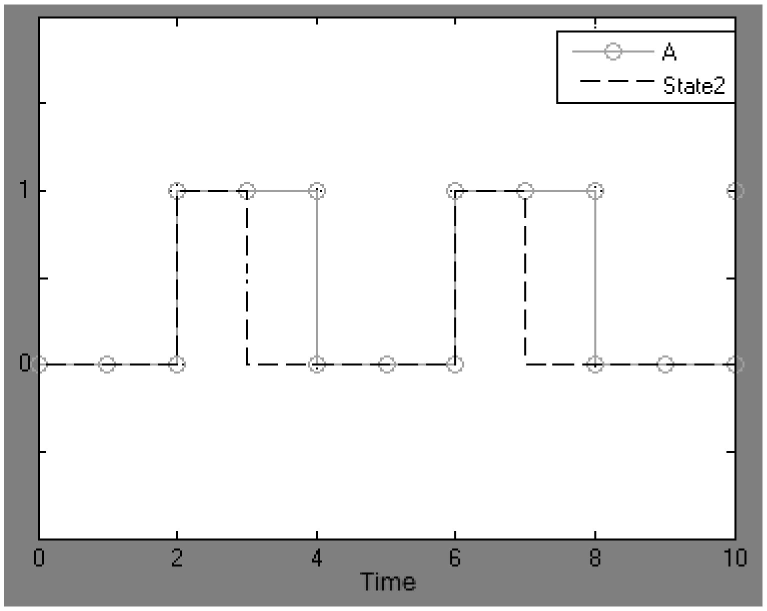


Figure 4. State2 Simulation Results for Transition Condition Example

Contrast these results with those of the event archetype. When B is defined as an input to Stateflow[*], there is no port on the left-hand side of the diagram. As seen in Figure 5, there is a top level port with a trigger symbol instead. Events are treated as instantaneous occurrences by Stateflow, only transpiring when the signal's value changes from non-positive to positive (i.e. rising) or from positive to non-positive (i.e. falling). This concept is similar to Simulink triggers, which cause subsystems to activate only when the trigger's value changes. Consequently, specifying an event to be an input to Stateflow causes the diagram to behave as a triggered subsystem. The contents of the diagram will only be evaluated when one or more of the input events change value. This invites the possibility of the diagram behaving in an asynchronous fashion regardless of the data rates of signals input to Stateflow on the left side.

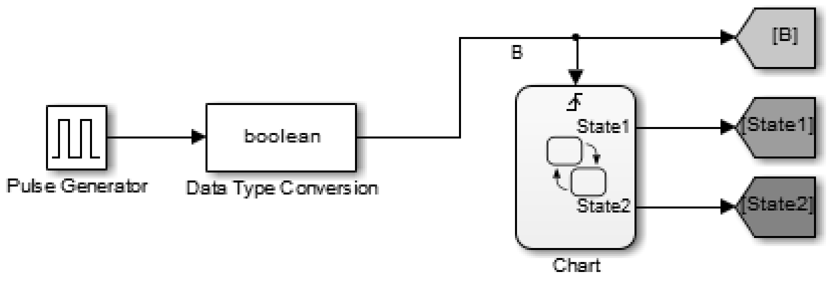


Figure 5. Event-Driven Transition Test Harness

Figure 6 displays the results from a simulation of the event-driven Stateflow diagram when the trigger condition for B is set to rising. We might expect a transition to State2 to take place two seconds into simulation when B is first set to true. However, since this is the first time B rises, this is also the first time the Stateflow diagram is activated. Prior to that time, the system is neither in State1 nor State2 as

[*] It is also possible to have events of local scope, meaning that they can be broadcast from one portion of the state machine to another. While this construct is worth exploring, it shall not be discussed here. See MAAB Style Guidelines section 8.3.2 for more information.[4]

can be confirmed in Figure 7.* Thus, we enter State1 two seconds into simulation and remain there at three seconds since there was no rise in B from the previous time step. We eventually enter State2 six seconds into simulation when the next rise occurs. Thus, we see that events are useful in situations where an instantaneous rise (or fall) in value is important, and when the developer desires Stateflow to behave as a triggered subsystem. Unless a watchdog timer is established to repeat at regular intervals, the diagram will only execute when an event transpires, potentially making the diagram behave aperiodically.

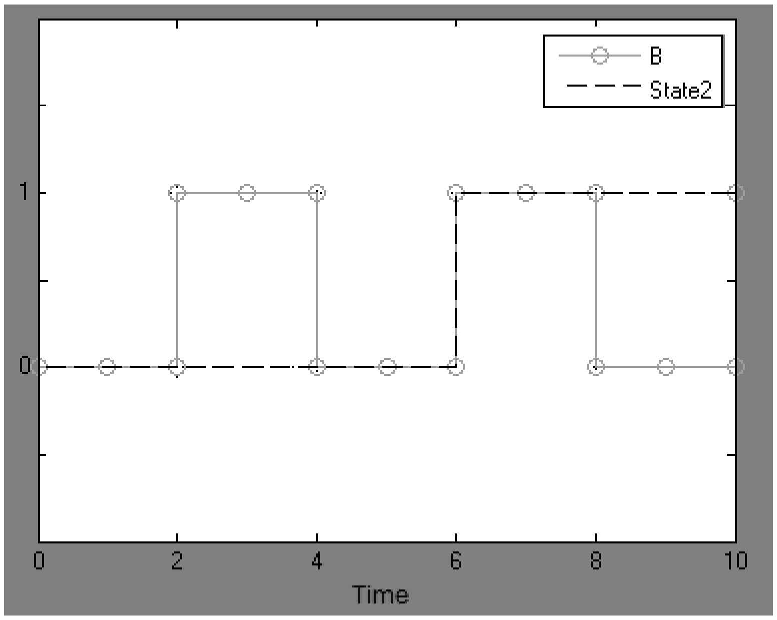


Figure 6. State2 Simulation Results for Event-Driven Example

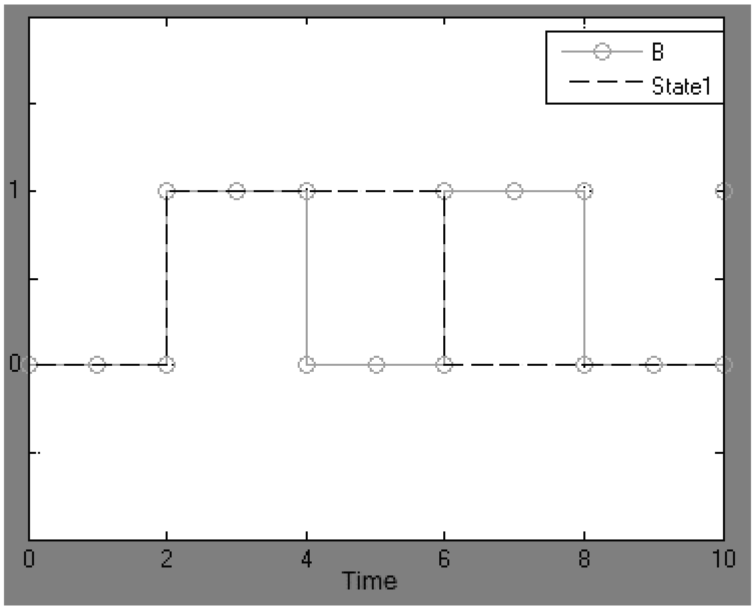


Figure 7. State1 Simulation Results for Event-Driven Example

* If the user prefers the first transition to State2 to take place at two seconds, then the diagram's properties can be amended to cause it to initialize at the start of simulation.[3]

Transition conditions and input events are permitted by both Mealy[5] and Moore[6] machine standards. The choice between the two styles is driven more by desired system behavior. If the algorithm is intended on executing at irregular intervals, then events are preferred. Railway track management is an example of an asynchronous application, as the software often remains inactive until a train approaches. Embedded vehicle software, by contrast, traditionally operates at a fixed rate. Such systems, therefore, tend to favor transition conditions. The MAAB[4] and Orion[7] style guides provide such advice on usage of events and transition conditions without explicitly recommending one or the other.

CHOOSING WHERE TO DEFINE ACTIONS

The operations performed when a criterion is satisfied within Stateflow are referred to as actions. Within a Stateflow state machine, there are three types of actions: state, condition, and transition. Figure 8 illustrates all three types. The declarations for the value of `y` within State1 and State2 are state actions. The statements entry or exit determine when the action takes place.[*] Declarations within the braces are condition actions; i.e., those taken when the condition A is satisfied. Transition actions are preceded by a forward slash and take place as the transition from one state to another transpires.

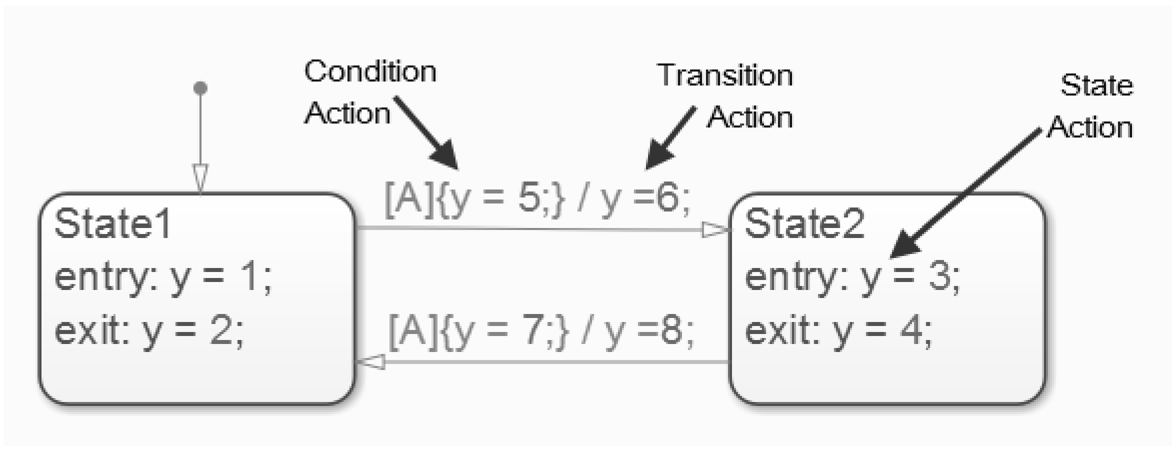


Figure 8. The Three Stateflow Actions

A simulation was run on the diagram from Figure 8 using a test harness identical to that shown in Figure 3. If we examine the results of this simulation (Figure 9), we see the value of `y` change from 1 to 3. When a transition from State1 to State2 is to occur, the following order of operation takes place, all on a single time step:

1) Condition A is satisfied, and condition action `y = 5` executed.
2) State1 is departed, and state exit action `y = 2` is executed.
3) Transition is traversed, and transition action `y = 6` is executed.
4) State2 is entered, and state entry action `y = 3` is executed.

Thus, we only observe the end result in the figure, `y = 3`. Examining generated C code won't confirm this behavior because the intermediate steps are recognized as having no functional bearing on the algorithm in our example and are optimized out. Only through reading the documentation or experimenting can the order of execution be revealed.

[*] There are other state actions such as `during`, `bind`, and `on` that will not be considered here.

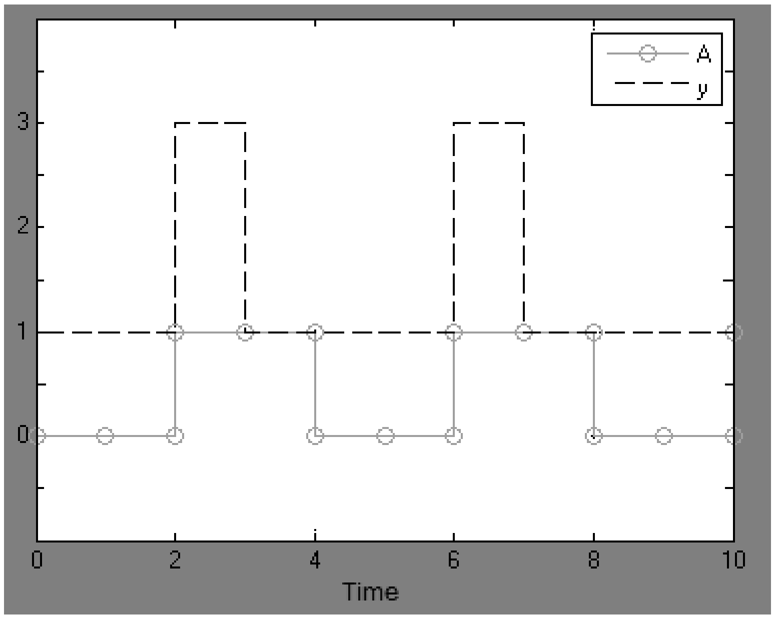


Figure 9. Simulation Results for Combined Actions Example

Were we to implement any one of these three techniques in isolation, they would be functionally equivalent in our example. We can conceive of other examples where this is not the case, as seen in Figure 10. The presence of a junction on the transition from State1 to State2 makes it such that two conditions are required simultaneously for transition to occur (both A and B). If only A is satisfied (B is held fixed at zero), then the condition action `y = 5` would be enacted whereas the transition action `y = 6` and exit action `y = 2` would not since no transition takes place in under these circumstances (Figure 11). Thus, choosing a condition or transition action has an impact on the algorithm.

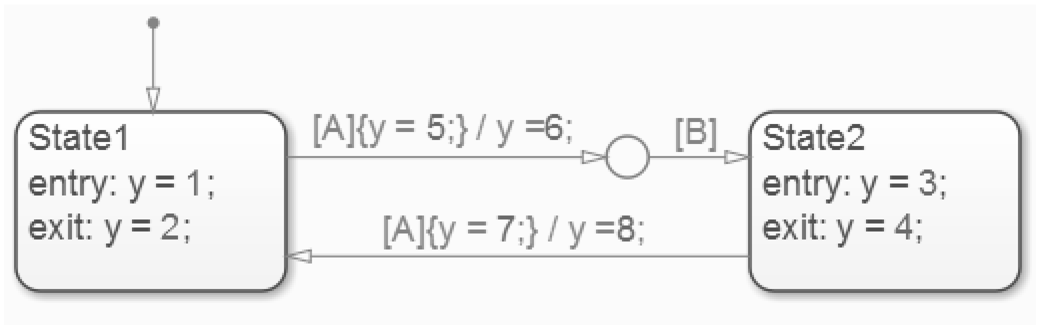


Figure 10. Example of Functional Distinction between Condition and Transition Actions

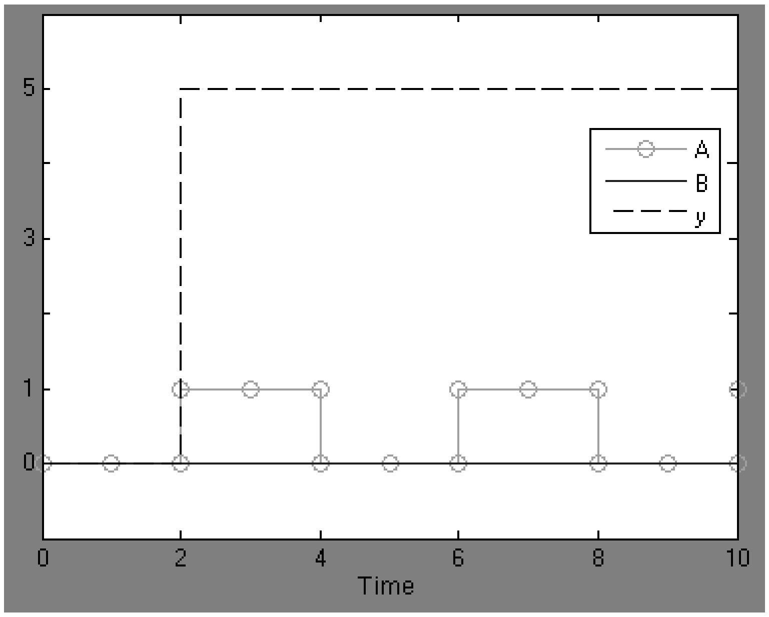


Figure 11. Simulation Results for Figure 10 Example

State actions, condition actions, and transition actions provide the user flexibility on when actions take place in relation to state transitions. While combining the constructs with junctions and feedback loops can facilitate more complex logical implementations, mixing them can result in unintended consequences. Because of this potential for confusion and the degree of redundancy between the options, Mealy standards specify that only condition actions are permitted. If adoption of Moore standards is preferred, only state actions are permitted.[8] Transition actions are forbidden in both approaches, but are allowed by the MAAB style guide (section 8.4.2) and Orion GN&C standards (section 4.5.4.2). No restrictions are placed on action usage in either document.

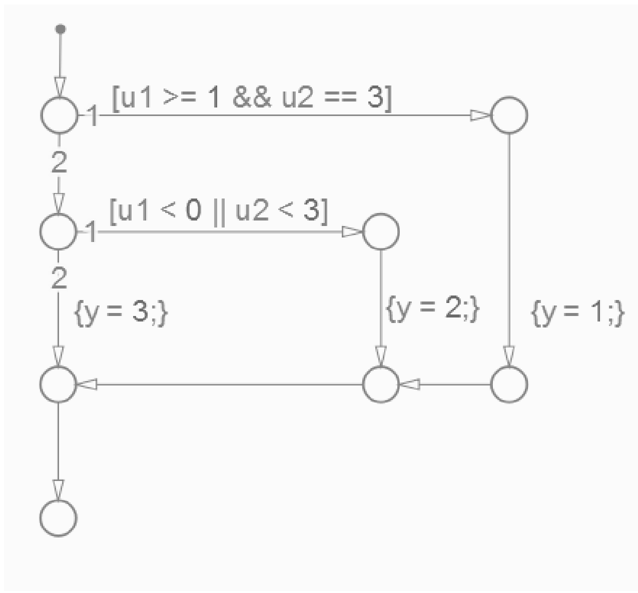
CHOOSING BETWEEN FUNCTION IMPLEMENTATIONS

Developers will often encounter situations where the algorithm governing a Stateflow condition or action exceeds what can be reasonably displayed on the state diagram. In such cases, a function can be defined elsewhere and accessed by the diagram. There are four common types of functions that Stateflow interfaces with.[*] Graphical functions rely on Stateflow semantics, junctions, and transitions to represent a flow graph. Truth tables may be used to specify actions to be taken for different combinations of logical expression values. MATLAB functions enable the user to write their algorithm in the MATLAB language. Finally, Simulink functions facilitate authoring algorithms with Simulink block diagrams.

For most algorithms, any of the four types of functions could be employed to represent the system. The question of superiority of one implementation over another then becomes open to interpretation. A common metric when given a choice of implementation in a Model-Based Design environment is the quality of the generated code. To that end, a simple if-elseif-else with two inputs and one output was

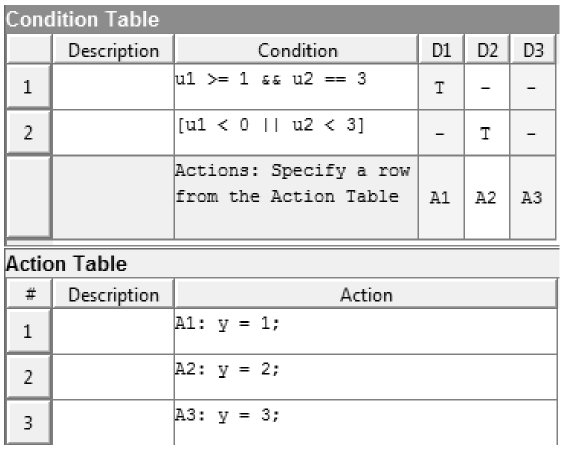
[*] It is also possible to directly interact with handwritten C functions when using C syntax as the action language. This can be a compelling technique for programmers versatile in C and/or working with an existing set of code.

designed as a graphical function, truth table, MATLAB function[*], and Simulink function. C code was generated from Embedded Coder in all four instances. The functions and their associated C code are displayed in Figures 12-15.[†]



```
    /* Graphical Function */
    if ((u1 >= 1.0) && (u2 == 3.0)) {
      y = 1.0;

    } else {
      if ((u1 < 0.0) || (u2 < 3.0)) {
        y = 2.0;

      } else {
        y = 3.0;
      }
    }

    return y;
  }
```

Figure 12. Graphical Function and Corresponding Generated C Code



Condition Table

	Description	Condition	D1	D2	D3
1		u1 >= 1 && u2 == 3	T	-	-
2		[u1 < 0 || u2 < 3]	-	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3

Action Table

#	Description	Action
1		A1: y = 1;
2		A2: y = 2;
3		A3: y = 3;

```
  /* Truth Table Function */
  if ((u1 >= 1.0) && (u2 == 3.0)) {
    y = 1.0;
  } else if ((u1 < 0.0) || (u2 < 3.0)) {
    y = 2.0;
  } else {
    y = 3.0;
  }

  return y;
}
```

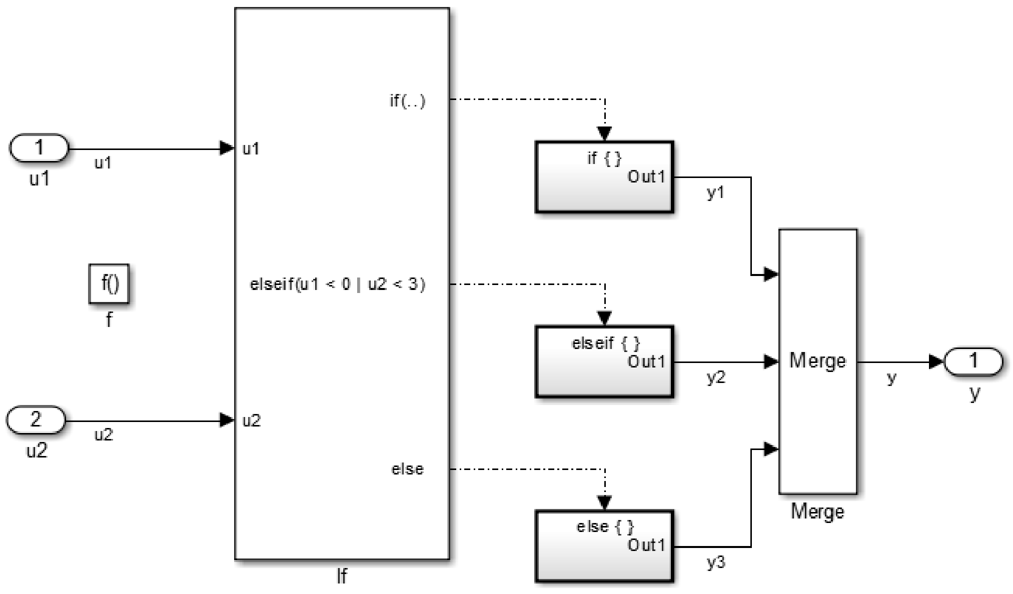
Figure 13. Truth Table and Corresponding Generated C Code

[*] There are multiple ways to invoke a MATLAB function within Stateflow. See the section "Choosing an Action Language" for further details.
[†] The developer has a great deal of flexibility in how a Simulink function is designed, even in this simple example. Different designs can result in different code implementations than the one displayed in Figure 15.

```
if u1 >= 1 && u2 == 3
    y = 1;
elseif u1 < 0 || u2 < 3
    y = 2;
else
    y = 3;
end
```

```
/* MATLAB Function  */
if ((u1 >= 1.0) && (u2 == 3.0)) {
  y = 1.0;
} else if ((u1 < 0.0) || (u2 < 3.0)) {
  y = 2.0;
} else {
  y = 3.0;
}

return y;
}
```

Figure 14. MATLAB Function and Corresponding Generated C Code



```
/* Simulink Function  */
if ((U.In1 >= 1.0) && (U.In2 == 3.0)) {
  B.y = 1.0;

} else if ((U.In1 < 0.0) || (U.In2 < 3.0)) {
  B.y = 2.0;

} else {
  B.y = 3.0;

}
```

Figure 15. Simulink Function and Corresponding Generated C Code

While this is only one case study, the results confirm that the C code is similar for all four examples. The graphical function produces a nested if-else whereas the other three examples produce an elseif. Variables are bundled into structures for the Simulink function and deviate slightly from the original nomenclature. This is due to the fact that, unlike the other three examples, code for the Simulink function is inlined in the main function of the Stateflow diagram, causing the variable names to adhere to root-level naming conventions. Thus, if being judged solely from the standpoint of generated code, graphical or MATLAB functions could be considered marginally preferable in this case study.

Mealy charts provide no restrictions on the type of functions that may be included in a Stateflow diagram. Moore charts, by contrast, forbid usage of all custom functions. The MAAB style guide recommends graphical functions, MATLAB functions, and truth tables for if-then-else constructs (sections 5.1.2 and 8.6.4). The decision to use or refrain from MATLAB functions is the left to individual companies (section 8.2.9), but the guidelines recommend them for situations with "complex equations" (section 8.6.4). Simulink functions are recommended for algorithms that utilize transfer functions, integrators, and lookup tables. Section 4.4.1.2 of the Orion GN&C guide provides recommendations on when to use Simulink, Stateflow, and MATLAB. It recommends Stateflow for logic-intensive algorithms and MATLAB or Simulink for numerically intensive algorithms.[8]

Commentary from the style guides is consistent with the notion that each type of function caters to certain classes of problems. Graphical functions can be useful for intricate, multilayered if-elseif-else constructs that might be difficult to follow as text written in MATLAB. Truth tables are compelling when a methodical examination of all possible combination of inputs is desired. It is the only option of the four that has built-in support for identifying missing or duplicate actions for condition combinations (i.e. underspecification and overspecification).[9] While certain developers may find flow graphs and truth tables intuitive, others might prefer MATLAB or Simulink functions based on their experience with those tools or the existence of intellectual property previously developed in those environments. In many cases, the expertise of the designer may dictate which path is going to result in the fastest development time while minimizing algorithm troubleshooting and maintenance.

CHOOSING AN ACTION LANGUAGE

Stateflow R2012b introduced a new block: the MATLAB Chart. With this block, developers express the rules of their finite-state machines and flow graphs via the MATLAB language. While certain semantics remain consistent (such as braces encapsulating condition actions), there are substantial differences with the C language syntax that has been part of Stateflow since its inception.[*] The action language is specified on a per-block basis, making it entirely possible to have a project that relies on a mixture of both types. However, for the sake of consistency, maintainability, and avoidance of design errors, it is recommended that a project adopt one action language and employ it throughout the Simulink model(s).[†]

Figures 16 and 17 display seemingly equivalent Stateflow diagrams expressed with MATLAB and C syntax as the action languages respectively. Users familiar with MATLAB will recognize use of parentheses to index into matrices, the index of the matrix starting at one, and percent signs demarking comments in Figure 16. C programmers will note bracketed, zero-based indexing[‡] and C syntax commentary in Figure 17. More subtle differences include the inability to directly cast a variable's type

[*] The C syntax used in Stateflow is similar to programming in C but not exactly like the C language in all respects.
[†] Both MATLAB and C will continue to be fully supported by MathWorks as Stateflow action languages.
[‡] This is the default behavior. The first index can be modified for individual variables to a non-zero value.

in the C syntax[*] and the lack of inline support for MATLAB functions supported by code generation[†]. The user may still utilize functions such as `cumsum`, but they must create an intermediary MATLAB function as shown in the figure. In both these cases, the code must be compatible with MATLAB Coder standards such that C code can be generated from it.[‡]

There are various other differences not visible in the figures.[3] Matrix math operations will generally be more easily achieved with MATLAB syntax. The C syntax adheres to the rules of scalar expansion, which prevents the user from creating functions with matrix inputs and scalar outputs. In addition, the size, type, and complexity of variables must be explicitly defined when opting for C syntax whereas these can often be inferred with the MATLAB syntax. When writing statements on transitions with MATLAB as the action language, transition conditions and condition actions can be distinguished and automatically encapsulated in brackets or braces. Other differences include support for C assignment operations (i.e., `y1++`) and the method of resolution when double precision numbers are added to non-doubles.

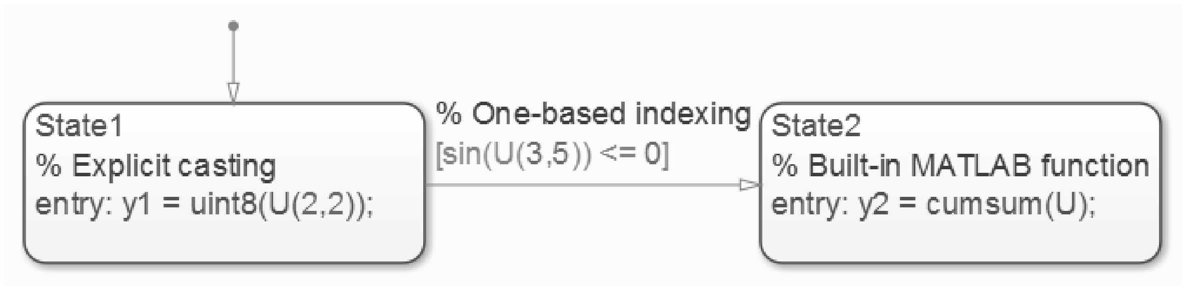


Figure 16. Example Stateflow Diagram with MATLAB as the Action Language

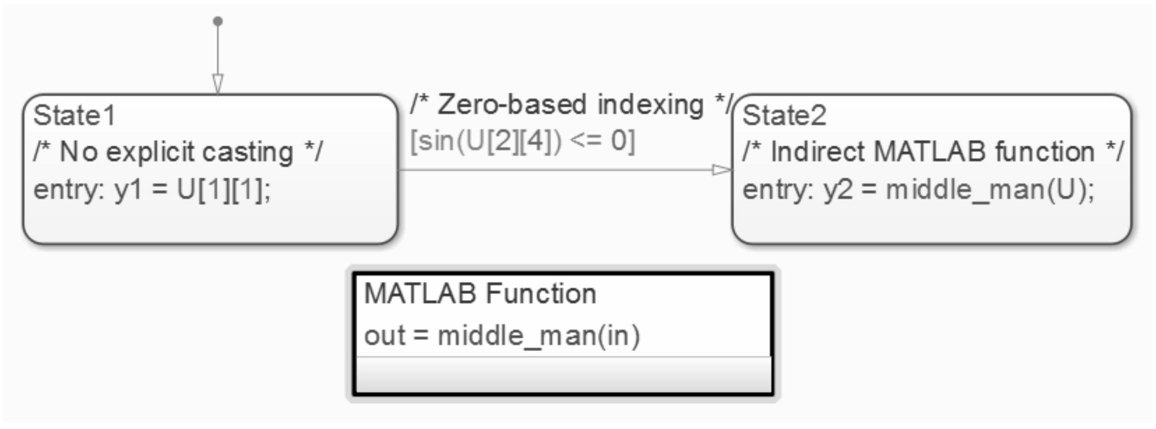
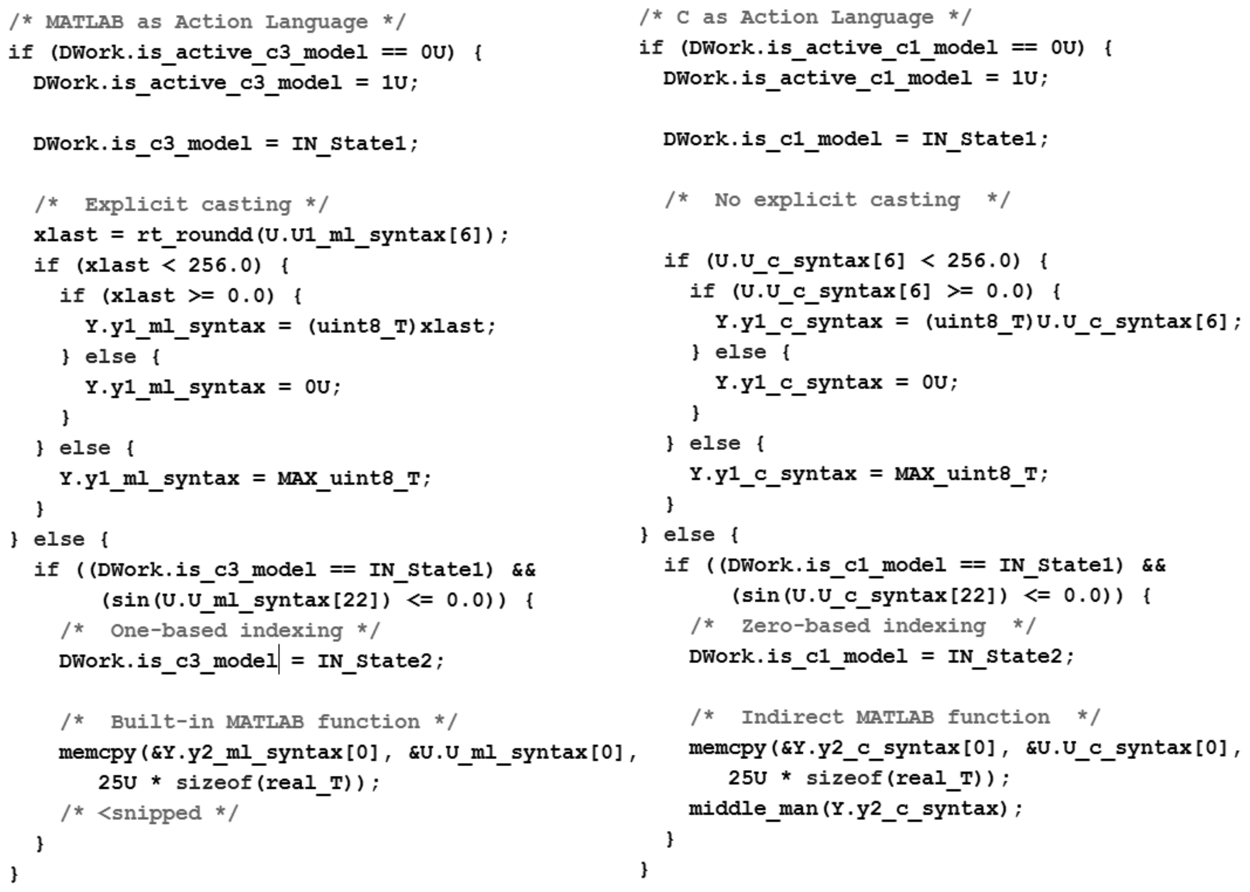

State1
/* No explicit casting */
entry: y1 = U[1][1];
/* Zero-based indexing */
[sin(U[2][4]) <= 0]
State2
/* Indirect MATLAB function */
entry: y2 = middle_man(U);
MATLAB Function
out = middle_man(in)

Figure 17. Example Stateflow Diagram with C Syntax as the Action Language

Generating C code provides insight into the precise implementation of both examples. Figure 18 displays a side-by-side comparison with the `cumsum` function (which had been inlined) removed from

[*] The user must specify the data type when defining the Stateflow data variable or use the `type` operator.
[†] MATLAB functions that are part of the standard `math.h` C library such as `sin` are supported by the C syntax.
[‡] When using C syntax, it is also possible to call a MATLAB function by writing `ml.<function name>`. While this permits the user to leverage the entire MATLAB language, it is not compatible with code generation and hence expressly forbidden by the MAAB Style Guidelines.[4]

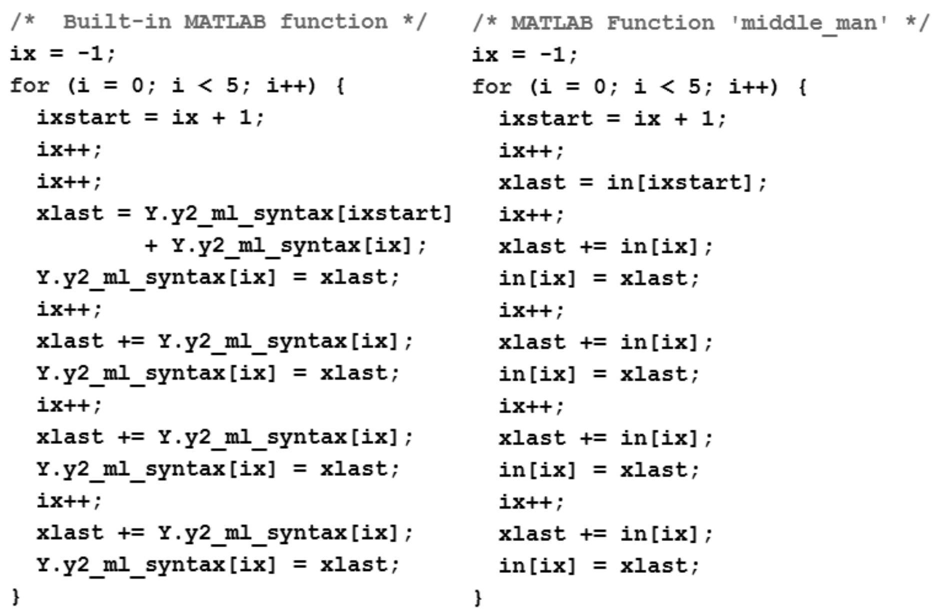
the MATLAB example. Figure 19 compares the C code produced from the cumulative sum function. Besides the variable names differing, they are identical. In Figure 18, there is an extra line of code and local variable in the MATLAB example related to the casting of `y1` as an unsigned 8 bit integer. Since MATLAB type casts round to the nearest whole number, and the function `rt_roundd` is included to ensure that the C code behaves in a similar manner. Casting to an integer in C rounds down (like the `floor` function), which is why the line is absent in the second example. It's a subtle difference, but the example illustrates how two seemingly identical Stateflow diagrams are not functionally equivalent due to differences between MATLAB and C.



```
/* MATLAB as Action Language */
if (DWork.is_active_c3_model == 0U) {
  DWork.is_active_c3_model = 1U;

  DWork.is_c3_model = IN_State1;

  /*  Explicit casting */
  xlast = rt_roundd(U.U1_ml_syntax[6]);
  if (xlast < 256.0) {
    if (xlast >= 0.0) {
      Y.y1_ml_syntax = (uint8_T)xlast;
    } else {
      Y.y1_ml_syntax = 0U;
    }
  } else {
    Y.y1_ml_syntax = MAX_uint8_T;
  }
} else {
  if ((DWork.is_c3_model == IN_State1) &&
       (sin(U.U_ml_syntax[22]) <= 0.0)) {
    /*  One-based indexing */
    DWork.is_c3_model = IN_State2;

    /*  Built-in MATLAB function */
    memcpy(&Y.y2_ml_syntax[0], &U.U_ml_syntax[0],
       25U * sizeof(real_T));
    /* <snipped */
  }
}
```

```
/* C as Action Language */
if (DWork.is_active_c1_model == 0U) {
  DWork.is_active_c1_model = 1U;

  DWork.is_c1_model = IN_State1;

  /*  No explicit casting  */

  if (U.U_c_syntax[6] < 256.0) {
    if (U.U_c_syntax[6] >= 0.0) {
      Y.y1_c_syntax = (uint8_T)U.U_c_syntax[6];
    } else {
      Y.y1_c_syntax = 0U;
    }
  } else {
    Y.y1_c_syntax = MAX_uint8_T;
  }
} else {
  if ((DWork.is_c1_model == IN_State1) &&
       (sin(U.U_c_syntax[22]) <= 0.0)) {
    /*  Zero-based indexing  */
    DWork.is_c1_model = IN_State2;

    /*  Indirect MATLAB function  */
    memcpy(&Y.y2_c_syntax[0], &U.U_c_syntax[0],
       25U * sizeof(real_T));
    middle_man(Y.y2_c_syntax);
  }
}
```

Figure 18. Generated C Code from Examples with C and MATLAB as the Action Language



```
/*  Built-in MATLAB function */   /* MATLAB Function 'middle_man' */
ix = -1;                          ix = -1;
for (i = 0; i < 5; i++) {         for (i = 0; i < 5; i++) {
  ixstart = ix + 1;                 ixstart = ix + 1;
  ix++;                             ix++;
  ix++;                             xlast = in[ixstart];
  xlast = Y.y2_ml_syntax[ixstart]   ix++;
        + Y.y2_ml_syntax[ix];       xlast += in[ix];
  Y.y2_ml_syntax[ix] = xlast;       in[ix] = xlast;
  ix++;                             ix++;
  xlast += Y.y2_ml_syntax[ix];      xlast += in[ix];
  Y.y2_ml_syntax[ix] = xlast;       in[ix] = xlast;
  ix++;                             ix++;
  xlast += Y.y2_ml_syntax[ix];      xlast += in[ix];
  Y.y2_ml_syntax[ix] = xlast;       in[ix] = xlast;
  ix++;                             ix++;
  xlast += Y.y2_ml_syntax[ix];      xlast += in[ix];
  Y.y2_ml_syntax[ix] = xlast;       in[ix] = xlast;
}                                 }
```

Figure 19. Generated C Code for Cumulative Sum

Even in cases where functional equivalence is achieved, there may be noticeable differences in the generated code depending on which action language is adopted. Certain advanced code generation options and verification and validation analyses may only be available with one of the two syntaxes. As a general rule, choosing C as the action language will provide deeper code generation and V&V capabilities due to its longstanding heritage.* Nevertheless, most users' needs will be satisfied with either approach, and any differences in capability will diminish in subsequent Stateflow releases. The decision of action language therefore becomes driven by organizational expertise. Teams with a stronger MATLAB background will prefer MATLAB as the action language whereas C programmers should opt for the alternative.

CONCLUSION

When adopting Stateflow for logic-intensive algorithm design, developers are faced with great deal of flexibility and choice in how those algorithms are implemented. Four fundamental choices are state transition definition, action definition, function representation, and action language selection. In all cases, the options available were found to cater to certain groups and applications. Event-driven state-machines are well suited for aperiodic and message-based applications whereas transition conditions work well for fixed-rate systems. State, condition, and transition actions have a set order of execution that should be understood when combining them together, relying heavily on junctions, or establishing state feedback loops. Graphical functions, truth tables, MATLAB functions, and Simulink functions offer similar design capabilities and code generation output; engineering judgment should dictate which approach offers the most intuitive manner to represent the precise algorithm in question. Similarly, MATLAB or C syntax should be employed as Stateflow's action language based upon the organization's expertise in each.

Examination of existing literature on state-machine standards can provide further guidance. Mealy machines rely exclusively on condition actions. Moore machines rely on state actions and forbid any kind of function in Stateflow. The MAAB Style and Orion GN&C guidelines provide further specifics on the usage of events and appropriate application of functions and truth tables. But every organization should draw their own conclusions as to the most efficient path forward in both the short-term and the long-term.

* The MATLAB syntax is so recent that neither the MAAB or Orion GN&C guidelines discuss the feature.

In some cases, one approach might provide a quick solution based on a team's current skillset, but investment in acquisition of new skills might lead to a more functional, scalable, maintainable, and robust system.

---

[1] Anthony, Mike and Friedman, Jon. *Model-Based Design for Large High-Integrity Systems: A Discussion on Model Architecture*. AUVSI's Unmanned Systems, San Diego, CA: Association for Unmanned Vehicle Systems International, 2008.

[2] Anthony, Mike and Behr, Matt. *Model-Based Design for Large High-Integrity Systems: A Discussion on Data Modeling and Management*. AAS GN&C Conference, Breckenridge, CO: American Astronautical Society, 2010.

[3] *Stateflow Documentation*. MathWorks Documentation Center. The MathWorks, Inc. September 2012.

[4] Donald, Hank. *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow Version 3.0*. MathWorks Automotive Advisory Board, August 2012.

[5] Mealy, George. *A Method for Synthesizing Sequential Circuits*. Bell System Technical Journal, volume 34. Bell System, 1955.

[6] Moore, Edward. *Gedanken-experiments on Sequential Machines*. Princeton, NJ. Princeton University Press, 1956.

[7] Henry, Joel. *Orion GN&C MATLAB/Simulink Standards*. FltDyn-CEV-08-148. National Aeronautics and Space Administration, October 2011.

[8] Anthony, Mike; Campbell, Will; and Petteys, Becky. *Model-Based Design for Large High-Integrity Systems: A Discussion on Logic-Intensive Algorithms*. AAS GN&C Conference, Breckenridge, CO: American Astronautical Society, 2013.

[9] Aberg, Rob. *Logic Design Using Stateflow Truth Tables*. The MathWorks, Inc, June 2004.