AAS 13-004

Model-Based Design for Large High-Integrity Systems: A Discussion on Logic-Intensive Algorithms

Mike Anthony, Will Campbell, and Becky Petteys

MathWorks

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

MODEL-BASED DESIGN FOR LARGE HIGH-INTEGRITY SYSTEMS: A DISCUSSION ON LOGIC-INTENSIVE ALGORITHMS

Mike Anthony[*], Will Campbell[†], and Becky Petteys[‡]

A large portion of the embedded software found in today's vehicles, be they sea-based, land-based, aircraft, or spacecraft, falls into the category of logic-intensive algorithms. The use of state machines has long been a common modeling practice for logic-intensive algorithms. As an abstraction of decision making procedures, finite state machines have always been an important construct in software engineering. Furthermore, it is hypothesized that the use of state machines with constrained semantics and deterministic behavior is critical for the development and verification of high-integrity applications.

This hypothesis is examined by comparing several different Model-Based Design approaches for the development of logic-intensive algorithms in a high-integrity environment. The goal is to understand the tradeoffs of several different modeling approaches at each step of a high-integrity workflow. The modeling approaches examined are: MATLAB®, Simulink®, Stateflow® using a subset of Classic semantics, Stateflow using Mealy[1] semantics, and Stateflow using Moore[2] semantics. These modeling approaches are compared at each step of a sample high-integrity software development workflow. This provides an opportunity to comprehensively evaluate the merits of each approach for development, automatic code generation, and model and code verification and validation.

The evaluation concludes that each approach is valid and provides significant benefit in at least one step of the workflow. As such, it is important that the goals and process requirements for a project be well understood before making a decision on which approach is optimal. However, within the context of the sample high-integrity workflow discussed in this paper, the use of Stateflow using Classic, Mealy, or Moore semantics can achieve optimal results.

INTRODUCTION

The past several years have seen a significant shift in the aerospace industry to more model-based engineering. However, the concept of using models to understand complex problems pre-dates this current trend. The use of models for logic-intensive algorithms has been a fundamental part of theoretical computer science for decades. Specifically, finite-state machines (otherwise known as finite-state automata or state machines) have long been a favorite modeling construct for representing complex decision logic.

---

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

Given this paper's focus on the use of state machines to model logic-intensive algorithms, it is useful to take a brief look at the basics of state machines. Fundamentally, state machines are an abstraction of decision making procedures. A state machine consists of states and transitions. A state describes where in a decision making process the machine exists at a given time. A transition is a decision path that allows a machine to go from its current state to another state. In a true finite-state machine, only a single state can be active at any given time. A simple example of a state machine constructed using Stateflow* is as follows:

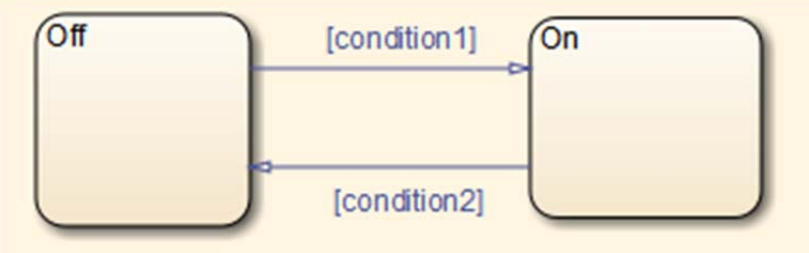

Figure 1: Simple State Machine

In a state machine, knowledge of the active state is important so an appropriate action can be taken. For example, if a switch is off, one may want to turn it on. Likewise, if a switch is on, one may want to turn it off, but there is no need to turn it on. These actions are specific to the state of the switch and can be performed either within the states, as in Figure 2, or during the execution of transitions, as in Figure 3.[3]

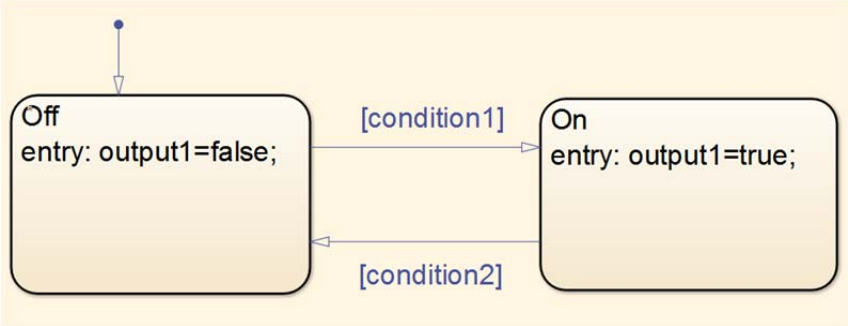

Figure 2: Actions in States

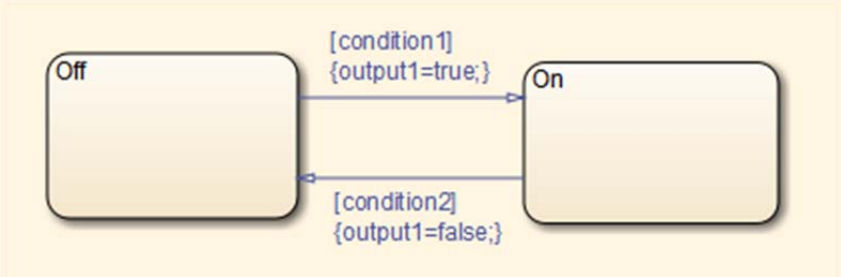

Figure 3: Condition Actions on Transitions

* Unless otherwise noted, all models, simulation results, and automatically generated code shown in this paper were created using MATLAB, Simulink, Stateflow, Simulink Verification and Validation, Simulink Design Verifier, MATLAB Coder, Simulink Coder, and Embedded Coder, release R2012b.

This raises the question of whether state-based actions or transition-based actions are the preferable choice. In a Stateflow chart using Classic semantics, both state-based actions and transition-based actions can be combined. However, in high-integrity systems, this kind of flexibility is not necessarily desirable and a more constrained and formal language to describe state machines is preferred. So, to constrain Classic semantics in Stateflow, it is strongly recommended to choose one of the above methods and enforce the selected method through a rigorous modeling standard. Another possibility is to make use of state machines with more formal semantics like Mealy machines[1] and Moore machines[2] in Stateflow. This will be discussed in the *Modeling Guidelines* section.

MODEL-BASED DESIGN FOR HIGH-INTEGRITY SOFTWARE DEVELOPMENT

With the recent shift in the aerospace industry to model-based engineering, it is interesting to examine how state machines are a valuable modeling language. Specifically, this paper will examine their value in a high-integrity software development process.

A high-integrity software development process requires a strict set of verification and validation procedures. The paper *Model-Based Design for Large High-Integrity Systems: A discussion on Verification and Validation*[4] offers a detailed discussion of a high-integrity software development process and the merits of using Model-Based Design to accomplish such a task. This discussion will not be repeated here. However, this example workflow is important to have in mind when discussing possible modeling techniques for logic-intensive algorithms. This workflow is a clear reminder that the creation of the design is only a single step in a rigorous software development process. When creating the design, it is important to understand the impact of design decisions on the rest of the process. This often creates a situation where the engineering team must make tradeoffs between the time and effort required to: develop the requirements for the system, develop a design that satisfies these requirements, implement this design in embedded software, and analyze and test this implementation.

Logic-intensive algorithm development is a particularly interesting example of these process-related tradeoffs. As previously discussed, state machines are a long-standing and accepted modeling technique for logic-intensive algorithms. Despite this precedence, state machines are often not used as the expression of the design in favor of a code-based approach. Typically in such situations, the group responsible for developing the design consists mainly of software engineers with many years of experience writing embedded software by hand. Sometimes, there is a preference in such groups to develop and express the design in code due to their familiarity with that form. In some cases, this is a perfectly acceptable practice and the choice between modeling with state machines and a code-based approach is a matter of personal preference. In other cases, the choice of a code-based approach is made in order to avoid the learning curve and cost associated with the team becoming proficient with a new, more advantageous, Model-Based Design approach using Stateflow as the modeling language.

There is a trend in the aerospace industry to quantify the value of a Model-Based Design process compared to a traditional development process[5]. The goals of this activity are to justify the investment in Model-Based Design tools as well as to reflect the efficiency gained by the use of these tools in cost estimation activities aimed at winning new business.

Figure 4 illustrates estimated return-on-investment (ROI) results comparing Model-Based Design to a traditional software development process using an ROI framework. This example uses averaged cost and efficiency data based on real customer data. The framework provides the flexibility to reflect the cost associated with adoption of new tools and the learning curve associated with these tools.[5]

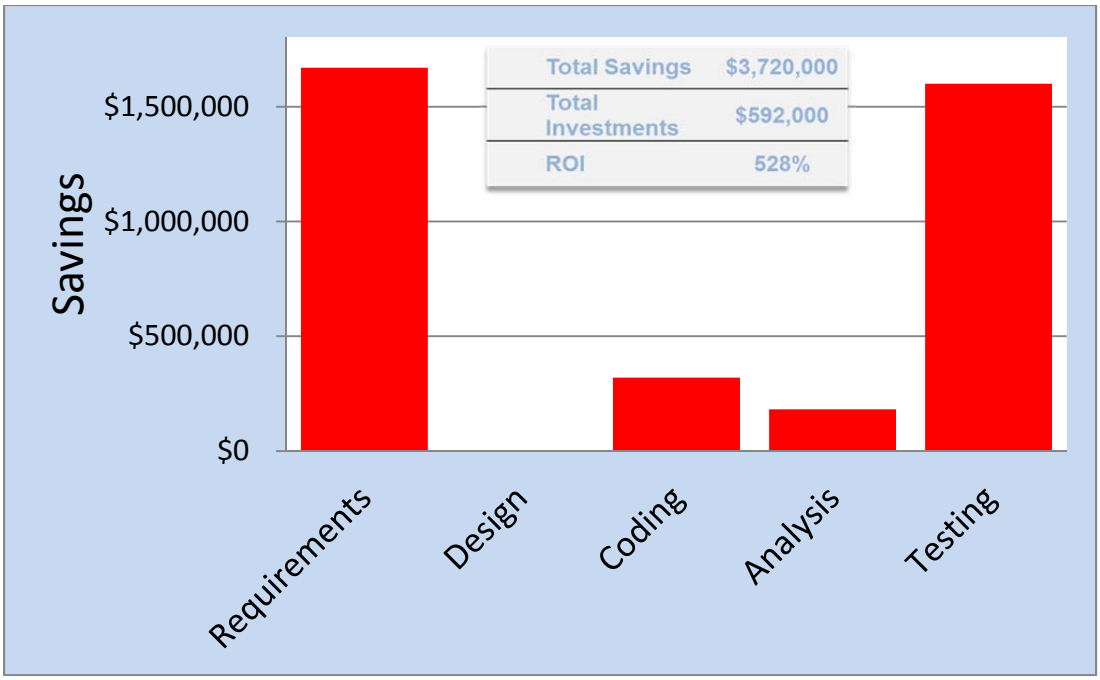

Figure 4: Model-Based Design ROI[5]

This data assumes a software development process that includes authoring requirements, developing a design, implementing the design in software, and analyzing and testing the embedded software. The significant savings realized in the Requirements development phase results from the use of an executable specification like a Simulink model and/or Stateflow charts. In Model-Based Design, the ability to execute the design allows early detection of errors in the requirements, such as missing requirements, inconsistent requirements, missing test cases for requirements, and unintended functionality in the design. In a traditional software development process, these errors are not usually detected until the Testing phase. The significant value of Model-Based Design shown in the Testing phase is a result of the ability to quickly diagnose and resolve errors, automate testing, and re-use test cases to test the model, code, and code running on a target processor from a single environment.

The ROI data illustrates that, in some cases, opting for a code-based approach rather than Model-Based Design for reasons of familiarity or fear of the learning curve of a new tool may prove a costly decision in the end. In order to understand the true cost of such a decision, the time and effort saved by opting for a code-based approach must be evaluated against the costs of developing requirements, implementing the design, and analyzing and testing the implementation. In the ROI data in Figure 4, adopting a new tool and overcoming the learning curve associated with it is included in the cost estimates of the Design phase. This is one of several contributors to why, in this example, there is no cost savings in the Design phase when using Model-Based Design compared to a traditional software development process. In some instances, the Design phase becomes more expensive when using Model-Based Design. However, the value of Model-Based Design over the entire process easily offsets any additional costs in the Design phase associated with new tool adoption and its associated learning curve. Also, the ROI for subsequent projects grows as the team gets more proficient with modeling and reuse of models.

Any ROI data employs many assumptions. As such, the data shown above may not be applicable to all applications and processes. The value of Model-Based Design is very much a function of the application, the team members, and the process. However, a safe generality is that

the more rigorous the process requirements for verification and validation, the more value Model-Based Design provides.

MODELING LANGUAGES

Having sufficiently motivated the use of Model-Based Design, the next step is to choose the optimal language for modeling. There are many different modeling languages from which to choose. Some are text or code based like MATLAB while others are graphical like Simulink. This paper assumes the reader has a working knowledge of MATLAB and Simulink. Previous papers in this series (Reference 6 and Reference 7) discuss the use of Simulink for high-integrity software development in great detail.

Recalling the previous ROI discussion, the majority of the value of Model-Based Design is related to the benefits achieved in the verification and validation parts of the development process. Note that development speed and familiarity with a modeling language is far less important than the ability to use the modeling language to assist with the verification and validation activities required in the development process.

Figure 5 shows an example of a High-Integrity Software Development Workflow similar to one discussed in a previous paper[4].

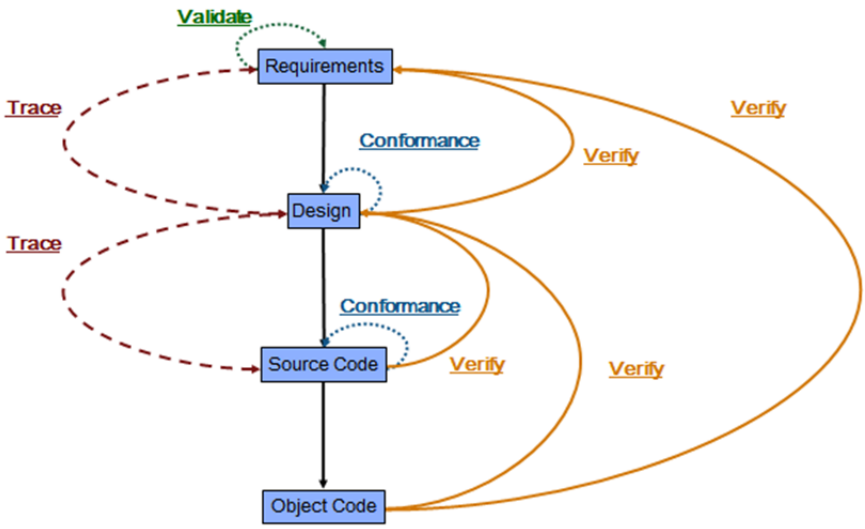

Figure 5: Example High-Integrity Software Development Workflow

It is a useful exercise to quickly compare modeling approaches and evaluate their benefits in a development process with rigorous verification and validation requirements. In this case, the modeling approaches to be discussed are: MATLAB, Simulink, and Stateflow. The following sections discuss any advantages one of these languages may have over the others for each step of the example process illustrated in Figure 5. Some steps are omitted from the following conversation when there is no significant advantage for any of the modeling approaches.

It should be noted that this paper does not compare Stateflow using the C action language and using the MATLAB action language.[3] The choice of action language in Stateflow is important and should be a key aspect of any modeling standard governing the use of Stateflow. However, any differences in functionality between the two action languages are not significant within the context of this paper. The examples in this paper make use of the C action language.

Development of the Design

In this step, a model is created that represents a design that satisfies the requirements. This is also the step where familiarity or experience with a particular modeling language is most beneficial. As previously discussed, the efficiency gained due to this familiarity must be weighed against the value provided by the modeling language in the other steps of this process. Regardless of the developer's familiarity with one modeling language there are distinct advantages to MATLAB or Stateflow over Simulink for complex logic. For documentation of the design, complex logic is generally agreed to be more easily understood when expressed in Stateflow or MATLAB code compared to Simulink. Complex logic tends to become verbose and more difficult to understand in Simulink when using fundamental constructs like logical operators and relational operators. Furthermore, the larger and more complex the algorithm logic gets, the more difficult it is to debug when using Simulink semantics. Because Stateflow provides better clarity and easier debugging than Simulink for complex logic, and provides all the same downstream capabilities as Simulink, the following sections will be sparse in their mention of Simulink as it is a design language better suited for complex mathematical algorithms rather than logic-intensive algorithms.

The choice between MATLAB and Stateflow typically comes down to a matter of personal preference and experience. For those familiar with state machine representations of complex logic, Stateflow provides a much clearer understanding of the algorithm. For those more familiar with writing code by hand, a series of `if-elseif` or `case` statements may be more familiar. That said, it can certainly be argued that when the logic becomes large and complex enough, even well-organized code can be difficult to understand due simply to the size and complexity of the algorithms being implemented. Furthermore, there are some instances for relatively simple algorithms where a state machine is a much simpler representation of the design. It is important not to downplay the importance of the documentation aspect of the modeling language. An example of such a situation is shown below.

Figure 6 shows a simple state machine modeled in Stateflow.

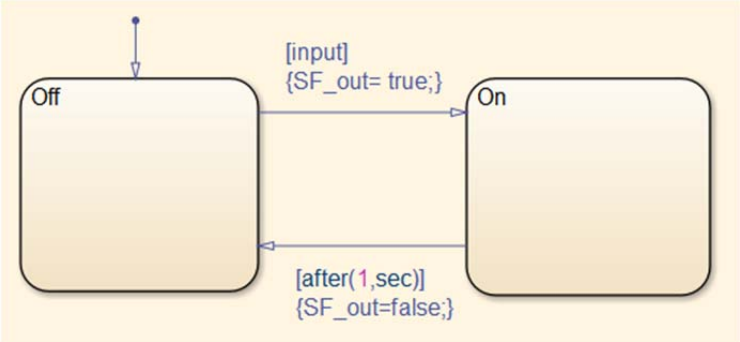

Figure 6: Stateflow Model of Temporal Logic

This state machine implements temporal logic, or logic dependent on time. In this case, the algorithm is intended to stay in the *On* state for 1 second and then transition back to the *Off* state. The equivalent MATLAB code for this algorithm is much more complex.

```
function ML_out = OnOff(input)
%#codegen
persistent count latch

if isempty(count)
    count = 0;
end
if isempty(latch)
    latch = false;
end

ts = 0.1;

if input
    latch = true;
end

if latch && (count < 1.0)
    count = count+ts;
else
    latch = false;
    count = 0;
end

ML_out = latch;
```

Figure 7: MATLAB Model of Temporal Logic

Even the most ardent MATLAB users would be hard-pressed to argue against the Stateflow model of this algorithm being significantly easier to understand. One of the most fundamental ideas behind Model-Based Design is that the model should exist as an executable specification. The value of the model existing as a document of the design is a significant part of the value of the model.[6]

Not only is the Stateflow model easier to understand, but is also easier to maintain. This simple example required two persistent variables in the MATLAB code. As the temporal logic grows more and more complex, the number of persistent variables necessary in the MATLAB code and the proper book-keeping of these variables can become difficult to manage and therefore more error-prone. In these situations, Stateflow offers a significant advantage.

Tracing the Design to the Requirements

A high-integrity process requires traceability from the design to the textual requirements to be bi-directional. Like Simulink, as a graphical modeling language Stateflow allows traceability to a very fine level of detail. Each individual modeling element can be linked to one or more textual requirements through functionality provided by Simulink Verification and Validation™. Not only is this important for the documentation aspect of traceability, but is also helpful when performing functional (requirements-based) testing of the model.

Conforming to Modeling Standards

The next step in the process is ensuring that the model conforms to a standard. In a Model-Based Design workflow, the modeling standard is inherently coupled with the coding standard. It is necessary to ensure that an appropriate modeling standard is defined and enforced such that the generated code also conforms to an appropriate coding standard.

Whether authoring MATLAB code or authoring Stateflow charts, a rigorous standard is very important. Stateflow, being an add-on to Simulink, takes advantage of the features of Simulink and Simulink Verification and Validation like the Model Advisor that assist with the definition of and automated checking against a modeling standard. There are several widely adopted standards for Simulink and Stateflow currently in use today. Examples of such standards include the MathWorks Automotive Advisory Board (MAAB) Style Guide[8] and the Orion GN&C Guidelines[9]. These style guides have two major focuses: readability (both of the model and the generated code) and functionality.

This results in an important choice for the engineer. In a high-integrity environment, opting to use MATLAB or Stateflow requires the authoring of a modeling standard to ensure consistency and traceability throughout the development process. Due to the flexibility of both MATLAB and Stateflow, an acceptable modeling standard can become quite lengthy and will almost certainly require the authoring of custom rules specific to a given application. Anything that can be done to automate the process of checking the model against the modeling standard will provide significant value over the life of a program. As such, if one modeling language supports the authoring of custom modeling rules and/or automated standards checking and one doesn't, that is a significant relative cost that much be considered when deciding between the two languages. Defining a modeling standard for the use of Stateflow in a high-integrity environment will be covered in the *Modeling Guidelines* section.

Verifying the Model against the Requirements

A key aspect of this step in the process is not only the execution of the requirements-based tests against the model, but also the measurement of model coverage resulting from the requirements-based tests. This coverage measurement is a key metric, illustrating the completeness and testability of the requirements, as well as the correctness of the design. As such, coverage and traceability are closely related. Very detailed traceability allows easier identification and debugging of missing coverage. As mentioned earlier, the level of detail with which traceability to requirements can be defined in Stateflow greatly assists with this task. Imagine a scenario where a complex state machine has been defined in Stateflow and, upon execution of the requirements-based tests, one of the states is never activated. If this state has been traced to a requirement, it is easier to determine why the test case for that requirement did not activate the state, if a requirement is missing, or if the state is unreachable. Failing to have such fidelity of traceability would make this task significantly more difficult. In a high-integrity environment, developing complex logic that allows both a high level of detail of traceability and coverage can be extremely costly and time consuming to do by hand. Anything that can be done to enable or assist with achieving 100% model coverage from requirements-based test is probably the most significant cost-benefit in a high-integrity workflow.

Automatic Code Generation

In Model-Based Design, automatic code generation and the modeling standard are tightly coupled. The modeling standard needs to ensure that the model executes, is readable, and can be integrated with other models. The modeling standard, with automatic code generation, is also the mechanism to enforce a coding standard on the generated code. Best practices for Stateflow usage

with code generation in mind is discussed in the *Modeling Guidelines* section, as well as in Reference 3.

Though code can be automatically generated directly from MATLAB, there is a much finer level of control over the generated code from Simulink and Stateflow. This is a natural consequence of the differences between graphical and code-based modeling languages. Despite the common misconception, generation of C code with MATLAB Coder® and Embedded Coder® from MATLAB code is not a simple line-by-line translation of syntax. It is necessary for Embedded Coder to take a much more global look at the MATLAB code in question in order for the generated C code to be functionally correct. Though this is necessary to help achieve functional equivalence, it does not always offer the traceability required by a high-integrity process. Stateflow can be very concise and traceable through the code generation process, depending on the modeling standard in place.

For example, imagine a Stateflow chart using Classic semantics and a modeling standard that allows the use of MATLAB functions within Stateflow. If this chart contains MATLAB functions that are very long and complex, the traceability through code generation is likely no better or worse than it would be compared to authoring MATLAB code alone. However, if the Stateflow chart uses a modeling standard that does not allow the use of MATLAB functions, then the traceability through code generation is likely to be better than from MATLAB code alone. This statement makes many assumptions about the algorithms, and as such should not be taken as an absolute truth. There are many cases where MATLAB and Stateflow are equivalent in this respect. However, there are also cases where Stateflow can provide an advantage. The engineer must analyze the tradeoffs of using MATLAB and Stateflow for their application to make the correct decision.

Tracing the Code to the Model and the Requirements

Like Simulink, one significant benefit of Stateflow during automatic code generation is an increased fidelity of traceability of the generated code to the model. Similar to tracing the Stateflow chart to textual requirements, each line of functional code generated by Embedded Coder from a Stateflow chart can be traced back to one or more modeling elements. Furthermore, if the states and transitions were linked to a textual requirements, then the resulting generated code will include the textual requirement as a comment. Therefore, Stateflow with Embedded Coder can provide traceability of every functional line of code to one or more textual requirements, assuming that the elements of the Stateflow chart were linked to textual requirements. This provides a significant advantage in a high-integrity process where it is typically required to trace every functional line of code to one or more requirements.

Similar to the discussion on traceability of the model to the requirements above, traceability and coverage are also extremely important at the code level. This is discussed in the *Verifying the Code against the Requirements* section.

Verifying the Source Code against the Model

In a high-integrity environment, verifying the source code against the model is most commonly accomplished through a code review. Manual review is the most common method of verification in this step due to the textual nature of the source code. The purpose is to show that all parts of the design were properly implemented in the source code. It is debatable if manual code reviews are actually sufficient to satisfy this goal; however, until recently, this was the only available method for this verification step.

New technologies in tools like Simulink Code Inspector™ are emerging to automate this task in some instances. Whether a particular modeling language supports the automation of this verification activity must be taken into consideration when determining the costs associated with the choice of modeling language. In a development process that requires this verification activity, anything that can be done to automate this activity and eliminate manual code reviews will result in significant cost savings over the life of the project.

Verifying the Code against the Requirements

In the section *Verifying the Model against the Requirements*, the importance of achieving 100% model coverage with requirements-based tests was discussed. Another reason why achieving model coverage is so valuable is that it eliminates the need to perform low-level testing of the code against the model. If the requirements-based test cases provide 100% model coverage, then verifying the code against the model and verifying the code against the requirements becomes the same activity.

The key capability for this step is the use of the Software-in-the-Loop (SIL) and/or Processor-in-the-Loop (PIL) capabilities of Embedded Coder. Using this capability enables the direct reuse of the requirements-based tests used to verify the model against the requirements. SIL and PIL are supported for both MATLAB and Stateflow. In addition, code coverage is an important measure of completeness in this step just as model coverage was a measure of completeness when verifying the model against the requirements. As before, coverage and traceability are closely coupled. When executing the test cases, it is important to also measure code coverage since model coverage and code coverage are not guaranteed to be equivalent. In the situation where the test cases provide 100% model coverage, but not 100% code coverage, it is essential to determine the cause of the discrepancy. After identifying the uncovered code, detailed traceability of the code to the model and the requirements enables faster root cause analysis and changes to resolve the model/code coverage discrepancy. The detailed traceability of Stateflow to the requirements in combination with the detailed traceability of the generated code to each modeling element in the Stateflow chart provides a significant advantage in this case.

MODELING STATE MACHINES IN STATEFLOW

There are instances where MATLAB and Stateflow each prove to be the optimal modeling language. In a high-integrity environment, however, the increased traceability in the development process offered by the use of Stateflow can prove to be advantageous over MATLAB in many cases.

What are the best practices for using Stateflow in a high-integrity environment? The first decision in the use of Stateflow is whether to implement Classic, Mealy, or Moore semantics. A brief introduction to Mealy and Moore machines may prove beneficial to this decision.

Mealy Machines

Mealy machines are state machines whose outputs are calculated based on the machine's current state and inputs. In a Mealy machine, the inputs are calculated before the outputs. At any given time step, if an input has changed, a dependent output can change on that same time step. Modeling a Mealy machine in Stateflow requires that actions can only happen on transitions, and not in states. More specifically within the Stateflow syntax, actions can only be defined as condition actions (the *{}* syntax). Figure 8 shows a Mealy machine in Stateflow.

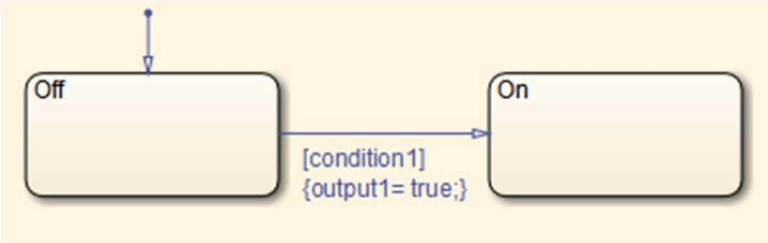

Figure 8: Mealy Machine in Stateflow

Using Mealy semantics and assuming the machine is in the `Off` state, if on a given time step, the input `condition1` changes from `false` to `true`, the output `output1` will be assigned the value of `true` on that same time step. Thus the output is a function of the state and the input.

Within Stateflow, it is important to note that when using Mealy semantics, there is no inherent limitation on how the transition condition is defined. Figure 9 shows a Mealy machine in Stateflow that uses a MATLAB function to determine the transition condition.

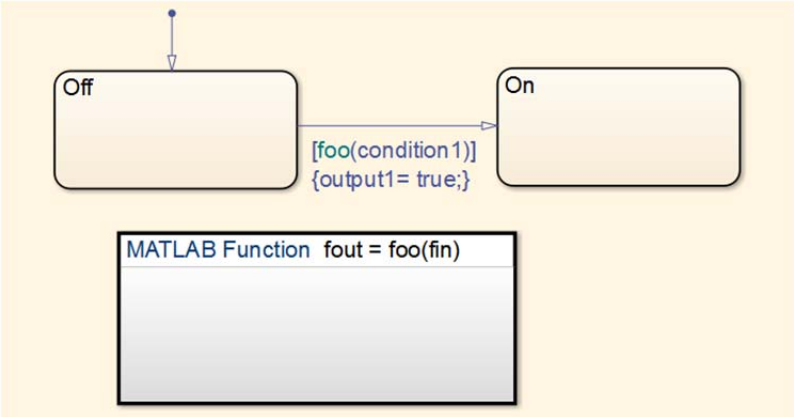

Figure 9: Stateflow Mealy Machine with MATLAB Function

This is perfectly acceptable within the rules of a Mealy machine and the majority of the MATLAB language is still available to author the function to determine the transition decision. Another example might be a Mealy machine using the MATLAB action language. In this case, any function on the MATLAB path can be called directly from within Stateflow. Both situations could be seen as lacking the language constraints necessary for a high-integrity environment. This illustrates that choosing to use Mealy semantics, though more formal than Classic semantics, is not alone sufficient for using Stateflow in a high-integrity environment. Whether using Classic or Mealy semantics, the development of a rigorous modeling standard to define proper usage of the features like MATLAB functions is still required.

Moore Machines

Moore machines are state machines whose outputs are calculated solely based on the machine's current state. Modeling a Moore machine in Stateflow requires that actions can only happen inside of states, and are not allowed on transitions. Furthermore, in a Moore machine, there is only a single state action as opposed to the `entry`, `during`, and `exit` actions available with classic Stateflow charts[10]. Figure 10 shows a Moore machine in Stateflow.

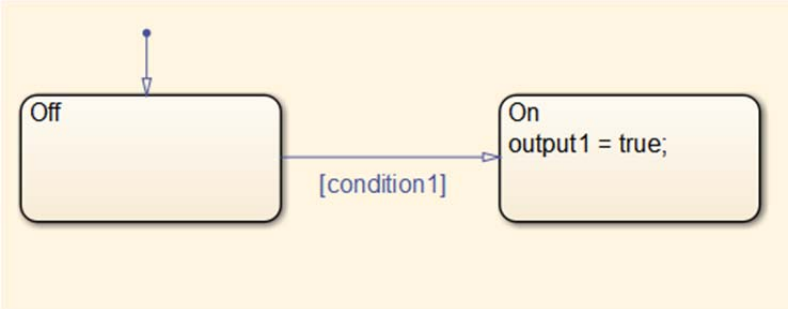

Figure 10: Moore Machine in Stateflow

Using Moore semantics, if on a given time step, the input `condition1` changes from `false` to `true`, the output will not change on that time step. In a Moore machine, at each time step the outputs are calculated first based solely on the current state. After the output calculation, inputs are then evaluated and, if necessary, a next state is determined. That next state will then become the current state on the next iteration of the Moore machine. In the example in Figure 10, that means that the output `output1` does not evaluate to true until the iteration after `condition1` goes true, the first iteration where `On` is the current state. This difference is illustrated in Figure 11, which shows the results of simulating the Mealy and Moore machines depicted in Figures 8 and 10, where the input `condition1` goes from false to true at time 0.1. The update rate of the simulation is 0.1 sec.

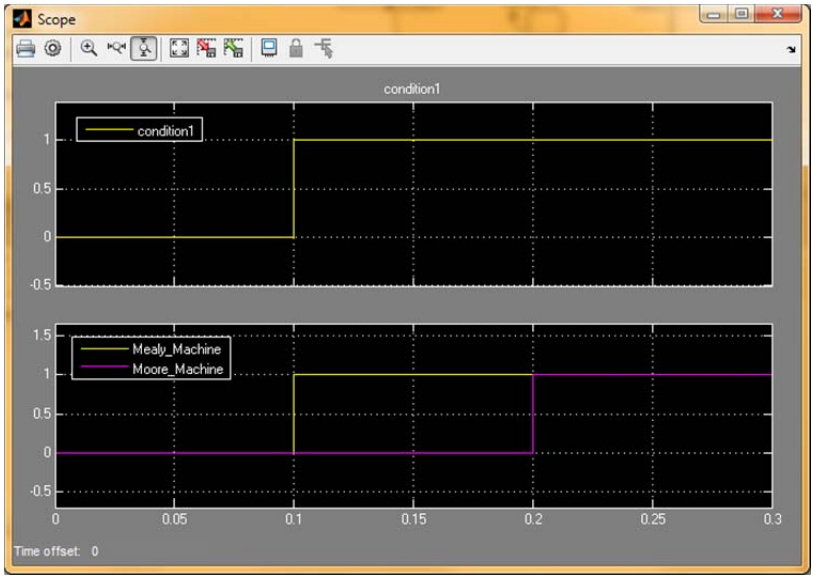

Figure 11: Simulation Results of Mealy and Moore Machines

In Stateflow, another difference between Moore semantics compared to Classic and Mealy semantics is that Moore semantics do not allow the use of any kind of custom functions. This means that Moore machines are restricted to the basic constructs of default transitions, states, and transitions. Actions can only be defined in states. The significant language constraints enforced in Moore machines can be advantageous for high-integrity applications. However, these language constraints may also prove too limiting to easily model some types of complex logic. The model developer will need to understand such implications before making an educated choice on whether Moore semantics are appropriate.

MODELING GUIDELINES

It is very important to note that when Stateflow is set to execute a chart as a Moore machine, it will enforce the Moore semantics and throw an error if the user attempts to define any behavior outside the strict set of rules that define a Moore machine. Likewise, if Stateflow is set to execute a chart as a Mealy machine, it will throw an error if the user attempts to define any behavior outside the strict set of rules for a Mealy machine. This inherent enforcement of the formal semantics of a Mealy or Moore machine may seem advantageous for use in high-integrity environments. However, this does not mean that Mealy or Moore semantics are the only acceptable use of Stateflow within a high-integrity environment. In fact, the most common approach is to use Classic Stateflow semantics. Regardless of the choice of Classic, Mealy, or Moore semantics, a rigorous set of modeling guidelines is necessary to ensure proper and safe usage of the available features.

A key practice mentioned in the discussion on Model Architecture[6] is to model in an intuitive manner. If there is a common expectation that the model or code generation process will behave in a certain way, the best practice is to ensure that the model conforms to that expectation. In this paper, modeling to conform to these common expectations shall be called the "intuitive rule." The intuitive rule represents a good thought process to determine an optimal modeling style for a given project. The intuitive rule applies to Stateflow as well as it applies to Simulink (see the Model Architecture Discussion[6] and Data Modeling and Management Discussion[7]).

There will necessarily be a wide spectrum of such modeling standards. At one extreme will be restrictive modeling standards that severely limit the available features for the developer in the name of increased consistency and formality, like the use of Moore semantics. Some modeling rules would still be necessary for considerations like readability; however, this is a small part of the modeling standards necessary when a broader set of the language is enabled. The other extreme will be modeling standards that allow the use of almost all of the available features of Stateflow. Allowing the use of all of these features will also necessarily require the development of rules to govern consistent uses of these features. The MAAB Style Guide[8] and Orion GN&C Guidelines[9] are examples of such modeling standard.

In general, the more features that are available to the developer, the more rules will be necessary to ensure safe usage of those features. This creates yet another tradeoff to consider when adopting Stateflow. A significant amount of time can be saved in the authoring and deployment of modeling standards for Stateflow if the available features are restricted. However, depending on the algorithms being modeled, such restrictions may prove counterproductive and allowing more features and spending time authoring a more thorough modeling standard may be the better choice in the long term.

With the breadth of functionality available within Stateflow, it's difficult to have a general and meaningful discussion about modeling rules. However, one example of the thought process behind creating modeling rules might prove educational.

Flow Chart Example

With any logic-intensive algorithm, constructs like `if-elseif` decisions are likely to be plentiful. There are several ways to implement an `if-elseif` construct in Stateflow[3]. This discussion will assume the use of a flow chart. Flow charts are Stateflow constructs comprised solely of transitions and junctions (i.e., no states), with conditions and actions defined on the transitions. Such a construct is incompatible with Moore semantics. However, it is acceptable when using Classic or Mealy semantics.

Within the constructs of embedded software, there are known rules to ensure safe implementations of such decision patterns. Figure 12 shows an `if-elseif` decision modeled in Stateflow.

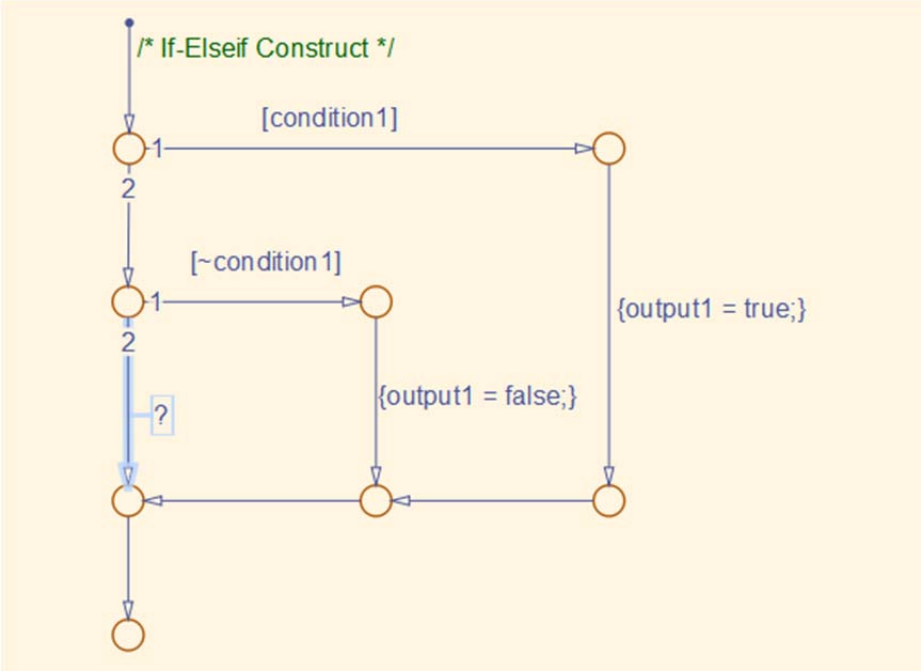

Figure 12: `if-elseif` Decision Modeled in Stateflow

This pattern was created using the Pattern Wizard in Stateflow. The textual requirements for this design would be:

1. If the input `condition1` is true, the output `output1` shall be true.
2. If the input `condition1` is false, the output `output1` shall be false.

The pattern in the Stateflow model may look a little curious at first. Specifically regarding the transition with the "?" near it.

Software engineers with experience in embedded software will likely understand why this transition exists. In fact, the MISRA AC AGC coding standard requires such a construct. Rule 14.10 states, "All `if`… `else if` constructs shall be terminated with an `else` clause."[11] In this example, the handwritten C code equivalent to the model in Figure 12 might look something like:

```
1  if(condition1)
2  {
3      output1 = 1;
4  }
5  else if(!condition1)
6  {
7      output1 = 0;
8  }
9  else{}
10
```

Figure 13: Handwritten C code `if-elseif` Decision

The transition in question represents the `else` statement in the above code. Although this conforms to Rule 14.10, this represents a serious problem in a high-integrity environment where 100% model and/or code coverage is required. In fact, this violates MISRA AC AGC Rule 14.1. Rule 14.1 states, "There shall be no unreachable code."[11]

Simulink Design Verifier™ can be used to detect dead logic in Simulink and Stateflow models via its Design Error Detection feature.[12] Running this analysis on the model in Figure 12 generates the results in Figure 14.

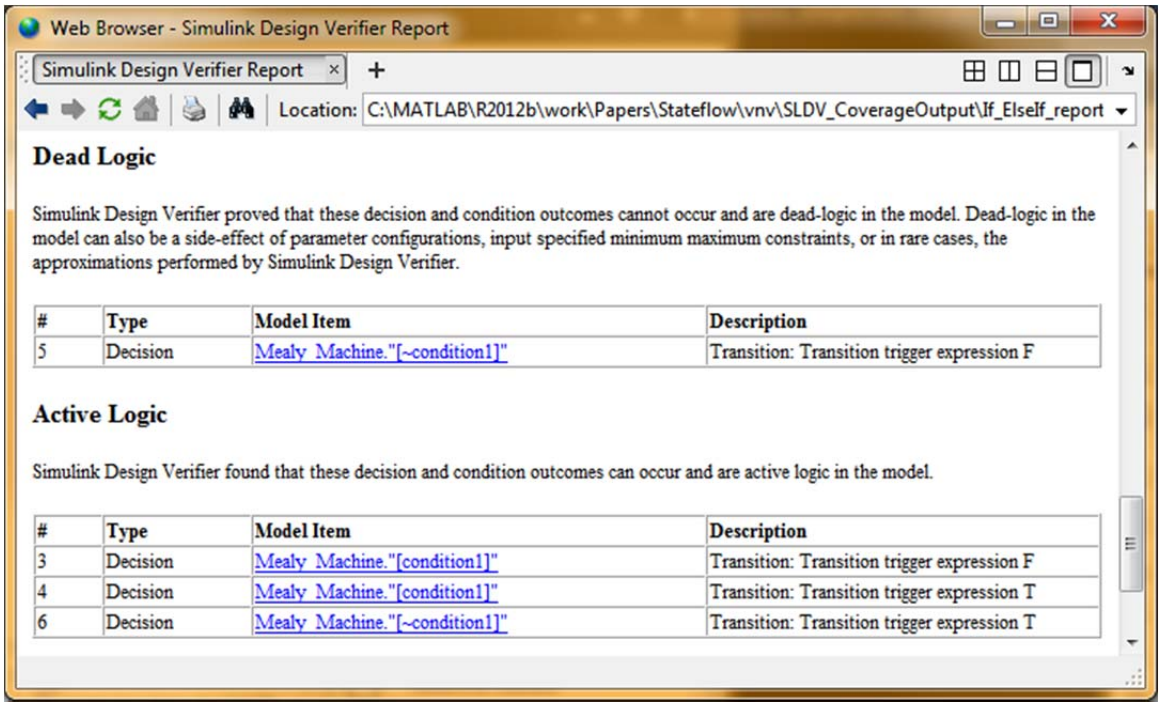

Figure 14: Results of Simulink Design Verifier Dead Logic Analysis for Model in Figure 12

This analysis reveals that it is mathematically impossible for the `elseif` statement of this logic to ever be false. A visual reexamination of the Stateflow model in Figure 12 confirms that the transition with the "?" can never be executed. To attempt to evaluate the `elseif` transition, the `if` transition must have been false (meaning `condition1` was `false`). If `condition1` has the value `false`, then the evaluation of the `elseif` transition must always be `true` (because `condition1` has to be `false` to even attempt this evaluation). Because the `elseif` condition can never evaluate to `false` this represents a decision coverage objective that cannot be met.

Given the flaw in the model in Figure 12, it is interesting to examine the code that Embedded Coder automatically generates from this model. Will the code have the same flaw as the model? How does the automatically generated code compare to the hand-written code in Figure 13?

```
 9   {
10     if (rtu_condition1) {
11       localB->output1 = TRUE;
12     } else {
13       localB->output1 = FALSE;
14     }
15   }
```

Figure 15: Embedded Coder Output for Stateflow Model in Figure 12

The automatically generated code in Figure 15 fails the intuitive rule because the decision is implemented as an `if-else` statement rather than an `if-elseif` statement. This use of the `if-else` construct in the generated code eliminates the coverage issue detected in the Stateflow model. The reality is that for this particular algorithm, an `if-else` decision is the proper construct to use rather than an `if-elseif` decision, as was used in the Stateflow model. Although it's convenient that Embedded Coder recognized this situation and optimized the automatically generated code, reliance on such automatic optimizations is unacceptable in a high-integrity environment.

However, following a proper high-integrity workflow, code should have never been automatically generated from this model. Instead, this error should have been detected during the verification of the model against the requirements. This is a perfect example of how Model-Based Design provides ROI. By using Stateflow and, in this example, Simulink Design Verifier, this error was detected long before any embedded software was written. In a traditional development process, it's likely that such an error would not be detected until much later in the process either through a time-consuming manual review of the code or during code coverage analysis.

Of course, detecting the error is only half of the problem. The other half is fixing the error. In Model-Based Design, the fix needs to be applied to the model. Figure 16 shows a Stateflow model where a more appropriate `if-else` decision structure is used.

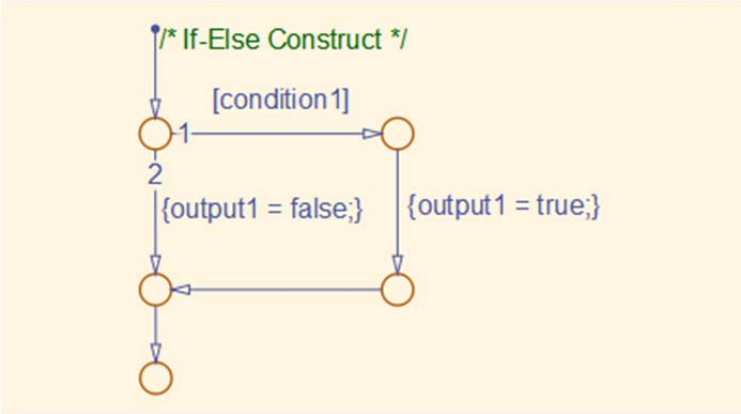

Figure 16: `if-else` Decision Modeled in Stateflow

Running Simulink Design Verifier: Design Error Detection on the Stateflow model in Figure 16 reveals no dead logic, as seen in the results in Figure 17.

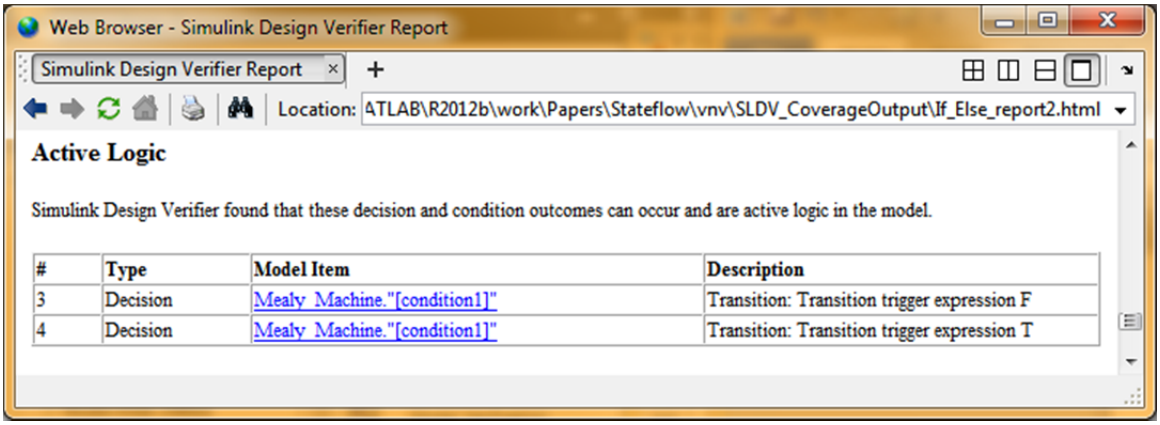

Figure 17: Results of Simulink Design Verifier Dead Logic Analysis for Model in Figure 16

The automatically generated code from the model in Figure 16 as follows:

```
 9   {
10     if (rtu_condition1) {
11       localB->output1 = TRUE;
12     } else {
13       localB->output1 = FALSE;
14     }
15   }
16
```

Figure 18: Embedded Coder Output for Stateflow Model in Figure 16

Note that this is exactly the same as the automatically generated code in Figure 15, which came from the model in Figure 12. However, this code and the model in Figure 16 follow the intuitive rule.

Going through this example has generated what is commonly referred to in industry as a "lesson-learned." A coding standard or modeling standard is really a collection of lessons-learned with the goal of reducing the likelihood of repeating a mistake in the future. This would be the perfect example of an opportunity to author a new modeling rule. In this case, the rule would state something like: when an `if-elseif` decision is modeled in Stateflow, make sure that an `if-else` construct is not the better algorithm. Implementing a rule like this in the Simulink Model Advisor could be written using MATLAB and take advantage of the Stateflow API. This MATLAB function would look for `if-elseif` constructs where the `if` and `elseif` conditions are logical opposites of each other (as was the case in the example above). In this situation, the Model Advisor rule would suggest the use of an `if-else` instead.

CONCLUSIONS

This paper discussed state machines as a longstanding modeling technique for logic-intensive algorithms and introduced Stateflow as a modeling environment for state machines. The value of modeling compared to hand coding software was then discussed motivating the use of Model-Based Design for high-integrity software development.

Within the framework of high-integrity software development, several modeling approaches for logic-intensive algorithms were evaluated. These modeling approaches included MATLAB, Simulink, and Stateflow. The evaluation determined any advantages that each modeling language has for use in a high-integrity software development process. This was accomplished by examining their value at each step of an example high-integrity software development process using Model-Based Design.

The comparison showed that for logic-intensive algorithms, Stateflow is the preferred choice of graphical modeling language compared to Simulink. The comparison of Stateflow against MATLAB was not as definitive. Analysis showed that the decision between these two modeling languages should be made on a case-by-case basis. The process for making this decision was discussed based on ROI calculations. The conclusion of this analysis was that opting for a particular modeling language solely due to familiarity with that modeling language is not always the optimal choice when looking at the ROI of the modeling language over the entire development process. Instead, the cost of the learning curve associated with adopting a new modeling language must be evaluated against the downstream value that new modeling language can provide.

Given the strong case for the use of Stateflow for modeling logic-intensive algorithms, this paper discussed the need for a rigorous modeling style for safe usage of Stateflow in a high-

integrity environment. The first conclusion was that regardless of the choice in Stateflow of Classic semantics or Mealy semantics, a rigorous modeling standard is required. Use of Moore semantics severely restricts the available language constructs in Stateflow meaning that the development of the modeling standard should be much easier as the formality of the language eliminates the need for guidelines for safe usage. However, such a restrictive subset of the language may not be suitable for modeling extremely complex logic. Thus, the model developer needs to study the tradeoff between the rigor of Moore semantics and the effort required to model complex logic when restricted to this subset of the Stateflow language.

This paper provided an example of the thought process for developing a modeling standard to optimize the use of Stateflow for a given application. The general conclusion of this discussion was two-fold. First, the primary concern of the modeling standard should be enforcing consistency in the modeling style across a large model. Second, developing the optimal modeling standard is best accomplished by examining the model and the generated code and evaluating both for their compatibility with model and code verification and validation activities including, but not limited to, conformance of the automatically generated code to existing coding standards.

For the most rigorous high-integrity software development processes, the use of Stateflow as the modeling language for logic-intensive algorithms often provides the maximum benefit over the entire development process. The inherent formality of Moore machines can be well suited for high-integrity software development processes. However, a modeling standard is still necessary even when using Moore semantics. In the situation that Moore semantics are too restrictive for the algorithms in question, the use of Classic or Mealy semantics in Stateflow is acceptable. In both cases, a rigorous modeling standard is necessary to ensure that the Stateflow models and the corresponding automatically generated code from Embedded Coder are compatible with the rigorous verification and validation processes associated with a high-integrity software development process.

[1] *A Method for Synthesizing Sequential Circuits*. **Mealy, George**. 1955. Bell Systems Technical Journal. Volume 34.

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

[3] *Understanding Model And Code Behavior For Stateflow Constructs*. **Campbell, Will, and Anthony, Mike and Petteys, Rebecca**. Breckenridge, CO: 13-031. American Astronautical Society.

[4] *Model-Based Design for Large High-Integrity Systems: A Discussion on Verification and Validation.* **Anthony, Mike and Behr, Matt, and Jardin, Matt, and Ruff, Richard.** Denver, C) : s.n., 2010. AUVSI Unmanned Systems.

[5] *Design Article: Measuring return on investment of model-based design.* **EE Times.** May 2011. http://www.eetimes.com/design/military-aerospace-design/4216303/Measuring-return-on-investment-of-model-based-design

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

[7] *Model-Based Design for Large High-Integrity Systems: A Discussion on Data Modeling and Management.* **Anthony, Mike and Behr, Matt**. Breckenridge, CO: 2010-9054. American Astronautical Society.

[8] *MathWorks Automotive Advisory Board Style Guide*. (2012). Retrieved December 14, 2012, from http://www.mathworks.com/automotive/standards/docs/MAAB_Style_Guideline_Version3p00_pdf.zip

[9] *Orion GN&C MATLAB/Simulink Standards*. (2012). Retrieved December 14, 2012, from http://www.mathworks.com/aerospace-defense/standards/FltDyn-CEV-08-148_MATLAB_Standards_v9_20111202.pdf

[10] *Supported Action Types for States and Transitions*. (2012). Retrieved December 6, 2012, from http://www.mathworks.com/help/stateflow/ug/supported-action-types-for-states-and-transitions.html#f0-128473

[11] MISRA AC AGC: Guidelines for the application of MISRA-C:2004 in the context of automatic code generation, Version 1.0. (Warwickshire: MIRA Limited, 2007) 30.

[12] *Dead Logic Detection*. (2012). Retrieved December 14, 2012, from http://www.mathworks.com/help/sldv/dead-logic-detection.html