Model-Based Design for Large High Integrity Systems: A Discussion Regarding Model Architecture

By Mike Anthony and Jon Friedman

MathWorks Inc, Natick, MA, 01760

INTRODUCTION

From complex controls problems like STOVL or VTOL manned aircraft, to fully autonomous systems, there is a definite growth trend in both the quantity and complexity of embedded software in today's aircraft systems. At the same time that the quantity of embedded software used in the aerospace industry is on the rise, so too is the required rigor with which this software must be developed. External software process certifications such as Federal Aviation Administration (FAA) DO-178B, previously required exclusively for commercial aviation projects, have greatly grown in influence. As a result, the importance of architecture has grown as the size and complexity of embedded software systems increases to meet these expanding requirements. Software architecture is typically defined as the hierarchical organization of the software necessary to accomplish the task at hand. Model-Based Design is well suited to support this task, as it extends the idea of a hierarchical organization further up the design process.

In a traditional design flow, requirements documents lead to algorithm development, which is then translated into code and embedded on the target processor. The software architecture in this case tends to be the first instance where hierarchy, order of computation, and interface dependencies between larger subsets of the software are defined. With Model-Based Design, these definitions can be made earlier in the process, such that the algorithms themselves are developed in a hierarchical manner. Using Model-Based Design, engineers can begin with a hierarchical organization in the model that exactly mimics the desired architecture of the target software environment. Accordingly, many of the architectural decisions for the project could be made earlier in the model environment.

This paper discusses the idea of model architecture and some of the factors to be considered when defining a model architecture. Establishing modeling practices for large models will address the ability for a large team to work on a model as well as simulation performance for a large model. The paper will also cover model architecture decisions for high integrity software development, including some stylistic considerations and considerations for configuration management and source control.

MODEL-BASED DESIGN

At the heart of Model-Based Design is the model, which is elaborated throughout the development process. The model provides an *executable specification*. "Executable" implies that the model exists as more than just a document, but as a functional part of the design process. As such, making the model functional is the first major influence on model architecture. This functionality is particularly important as the size and complexity of a model increases. With increasing size, this functionality includes organization to manage a large number of blocks as well as the execution performance of the simulation.

The “specification” in executable specification implies that the model, aside from being an environment in which an algorithm is developed and tested, serves as documentation of the algorithms. The documentation aspect of this idea has a distinct influence on the model architecture, to address readability, reusability, and annotation. Furthermore, as with other documents, the model must be compatible with configuration management and source control procedures determined by the process architecture. These considerations will also impact the model architecture.

The fundamental idea behind the architecture introduced in this paper is that the model should be a self-contained, fully specified, deterministic, hierarchical description of the algorithm. This idea impacts the tools used to componentize the model, how parameters are specified in the model, how signals are used in the model, and other modeling decisions. As with all engineering decisions, choosing a model architecture involves tradeoffs. The advantage to choosing a rigorous model architecture is the increased level of clarity and determinism in the model. The disadvantage is that working within such a model architecture requires a strict set of rules, greatly reduces the flexibility of the model, and significantly increases the required effort to complete the model. Thus, the algorithm developer must determine where the task at hand fits on the spectrum of modeling tasks. One side of this spectrum is modeling for rapid prototyping, where speed and flexibility are of great importance. The other end of the spectrum is modeling for a high integrity software development process, where adaptability is unacceptable, and determinism and repeatability are essential. For the most part, the suggestions contained herein are for tasks on the high integrity extreme of the spectrum, where to a certain extent an increase in effort is an acceptable tradeoff for rigor.

HIERARCHICAL COMPONENTIZATION AND MODULARIZATION

In the context of the model as an executable specification, model architecture is greatly influenced by the functional aspect of Model-Based Design. As with traditional software development, organization is necessary for understandability, clarity, reusability, and ability to debug. A truism for embedded systems is that when writing code, it is possible to write the entire necessary functionality for an embedded control system in a single contiguous line of code. However, this is also universally accepted as bad practice. Thus, code is broken into functions. The lowest-level functions perform the actual algorithms. Higher level functions create a hierarchy that determines the order of execution of these lower-level functions. The model environment should be subject to these same considerations. Similar to the single contiguous line of code, it is possible to fully define an immensely complex algorithm in a model via thousands of blocks with no hierarchy. However, this is also universally accepted as bad practice. For the same reasons as in the code environment, it is highly desirable to use subsystems like functions, to include some hierarchal organization of the functionality.

Like a function-call hierarchy, subsystems can be best thought of as a hierarchical organization tool, where all of the actual algorithmic functionality happens at the lowest levels. This hierarchical organization is perfectly suited to componentization and modularization of the model.

The terms componentization and modularization are often used interchangeably. However, in this particular case the definition of the terms will have a distinct difference. Componentization is the idea of subdividing a model into functional groupings. These functional groupings, or components, are exactly like functions in a code environment. Modularity relates to the level of isolation and/or interchangeability of a component. In a code environment, a highly modular component would be a reusable or reentrant function, with the function being completely self-contained and having no data dependencies beyond the specified inputs and outputs. Conversely, a less modular function would rely heavily on global data. The Simulink software environment has a number of constructs to support componentization and modularization of the model. Which constructs are most appropriate depends on model size and complexity, organizational influences, and required level of rigor. The following sections discuss the varying options for componentization, in order from least to most modular.

Virtual Subsystem

The first and most basic level of model componentization is the virtual subsystem. A virtual subsystem is simply that, a virtual boundary. Once a model update or Update Diagram (Ctrl-D in Simulink) is performed, these virtual boundaries disappear, essentially flattening the hierarchy of the model and putting all of the functionality at one level. Only after this flattening is the data-dependency sort for execution order performed, meaning that the virtual subsystem's graphical componentization of the model has absolutely no functional meaning, particularly in terms of block execution order. Though extremely useful for graphical organization, it is easy to imagine for models consisting of thousands of blocks how this construct alone might be problematic. Thus, for larger models some additional thought must be put into how the model is organized.

Atomic Subsystem

The next level of componentization, and the first with implications on modularity, is the atomic subsystem. Atomic subsystems allow not only graphical organization of the model, but also a functional componentization. The key difference is that atomic subsystems execute as a unit, and all functionality contained within that atomic subsystem will execute to completion before another part of the model is executed. Thus, unlike virtual subsystems, during a model update an atomic subsystem is treated as a functional group, exactly like a function in a code environment. This functional boundary allows the hierarchical organization in the model to also represent a functional hierarchy for simulation and for the generated code. In fact, atomic subsystems allow several different options when generating code, including the generation of reusable functions. For many applications this level of componentization is sufficient. However, when dealing with large simulations, other issues may arise.

Model Referencing

One of these issues is how to enable many people to work on a large model at the same time, without causing significant configuration management challenges. One way to address this issue is to find a way to modularize the model beyond the capability provided by atomic subsystems. Recalling the idea of the model as an executable specification, model referencing provides some significant advantages from the specification perspective. Model referencing is the idea of embedding an independent model file within another model. Model referencing requires that the external interfaces of the embedded model be fully defined such that the parent model can understand the input/output (I/O) structure and thus communicate with the referenced model. This definition means that at a minimum every signal's data type, size, and sample time must be defined a priori (i.e. not inherited). From a high integrity point of view, this I/O definition is not only highly desirable, but in most cases is required. Furthermore, model referencing also allows independent configuration control of the referenced models. Well-defined functional boundaries also allow many engineers to work on independent portions of the model without interfering with each other's work.

A second issue that can be addressed by model referencing is simulation performance. From a functional perspective, model referencing also provides significant advantages for simulation performance. A referenced model can be run in one of two modes. Normal mode executes the referenced model as a model, where the Simulink solvers operate on the blocks inside the model reference boundary. Accelerated mode generates code for the blocks within the referenced model, then compiles and interfaces this code with the rest of the parent model. Because execution of compiled code is inherently faster than the solver operating on Simulink blocks, model referencing operating in Accelerated mode can greatly increase simulation performance. However, there are two major costs for this simulation performance. The first cost is that the process of generating and compiling the code for a referenced model in Accelerated mode can significantly increase the time required to update a model. Thus, the trade is a longer update time in exchange for a faster simulation. The second cost is that the compiled code implemented in Accelerated mode does not offer the internal visibility available when executing Simulink blocks. This cost is addressed by Normal mode. However, in Normal mode, the simulation performance benefits are not realized. Furthermore, Normal Mode is still subject to the hard boundary definitions required by Model Reference, so any changes that require altering this interface can be more cumbersome than when using a subsystem implementation.

With this in mind, much thought must be put into choosing how to implement model referencing in a successful model architecture. The specification-related influences on model referencing want to push the model referencing boundaries down the model hierarchy to take advantage of the modularity provided. However, the functional-related influences want to push the model referencing boundaries up the model hierarchy to maximize simulation performance. These opposing influences coupled with the fact model referencing requires hard boundary definitions can cause confusion on how best to use model referencing.

To manage this conflict, the capability exists to switch from an atomic subsystem to a model reference, and vice versa. This capability can be used to move the model reference boundaries up and down the hierarchy depending on the task at hand. Thus, when dealing with configuration management, or the need for many team members to work on the model at the same time, the model referencing boundaries can be pushed to the lowest levels of the hierarchy to take advantage of the modularity provided by model reference. The algorithm designer can revert to subsystem boundaries for the part of the model requiring design and/or analysis where the interfaces may change, while using model reference for the rest of the model for increased simulation performance. For integration testing, this approach allows algorithm designers to optimize the model configuration based on the part of the model under consideration. Finally, for system-level testing or for code generation, these model referencing boundaries can be moved up the model hierarchy to maximize the performance benefits of model referencing in Accelerated mode.

Ultimately, model referencing boundaries should correspond to software component boundaries architected by software engineers for the target software environment. This does not mean that hierarchical model referencing is not suggested. In fact, for high integrity software development seeking certifications such as DO-178B Level A, using hierarchical model referencing can significantly reduce the amount of work involved for regression testing. In this type of environment, each time code is generated from a model it must be reviewed again and tested. Because the model reference represents a hard boundary, once the code from that referenced model is verified, treating it as a black-box function prevents the need to re-verify that function as a result of changes elsewhere in the model. Thus, even if a change is made up the model hierarchy requiring changes to the code that calls the model referencing function, this function itself need not be regenerated.

Libraries

Whenever the concept of model references is discussed, invariably the topic of libraries closely follows. Libraries are an extremely useful way to deal with block constructs that have many instances in a model as they allow the ability to link all of these instances to a single source. In doing so, the process of making a change to multi-instanced construct is simplified to altering the library source block only. The idea of a single source linked to many instances in the model seems to be a good place to start the application of configuration management and source control within a large model. However, as the above discussion suggested, model referencing and the modularity it provides is the proper tool for subdivision of a model for configuration management. So what is the most appropriate use of model referencing as opposed to libraries? At its absolute core, as with all of the topics discussed herein, this decision is ultimately based on the engineer's preference. That said, the discussion above provides a significant amount of insight into when to use libraries and when to use model referencing. Recall that a library is simply a subsystem. This subsystem has some additional properties in that it can be linked to some library version, but at its core it is still a subsystem. Thus, it provides none of the advantages achieved with model referencing with respect to modularity or simulation performance. It is quite common that libraries, rather than model referencing, are seen as the first step to apply some kind of configuration control within a large model. Interestingly, when not used carefully, libraries can negatively compromise the ability to configuration manage a large model. The ideas of modularity and configuration management go hand in hand.

As discussed above, libraries (being inherently subsystems) provide little capability for modularity. In fact, they provide the opposite capability when used hierarchically. Consider a hierarchy of linked libraries of considerable depth. An algorithm designer needs to make a change at the lowest level of this hierarchy. To do so, that designer must update the library block. However, because this block resides in a subsystem that is also a linked library, the parent subsystem has also changed and thus the library block must also be updated. This parent subsystem is also part of another subsystem going up the hierarchy, requiring that that parent subsystem's library block also be change. This cycle continues until you reach the top level of the hierarchy. In this architecture, making a change at the lowest level forces a change in every block in the hierarchy, including that low-level block. Furthermore, to make that initial change, the designer must break every library link in the hierarchy. This approach is neither modular nor conducive to configuration management. This is not to say, however, that libraries do not provide a very useful capability. In the case where an algorithm developer creates a functional grouping of blocks that will be used many times throughout the model, implementing this grouping as a library makes sense. Typically such a functional grouping would rarely require editing, thus avoiding the issues discussed above. However, if a change is ever required to this functional grouping, the change can be made in a single place (the library block) and this change will then propagate to all instances of this function in the model. Furthermore, this library block can be included in a user-defined library that appears in the Simulink library browser, allowing a team to essentially build a customized blockset for their specific application. Such an arrangement is highly useful. Thus, libraries, in general, are extremely useful for creating small reusable functional groupings. However, they do not offer any capability to modularize a model. It should be noted that the ability to place referenced models within a library provides some advantages with respect to configuration management. This is a topic that will be covered in depth in a different paper specifically on this topic.

CONTROLLING EXECUTION ORDER

The above discussion showed that the ideal model hierarchy includes a combination of atomic subsystems, referenced models, and libraries depending on the rigor of the software development process. By opting for one of the more modular approaches, the algorithm designer procures the additional benefit of being able to begin to specify the order of execution of the algorithms as expressed in the model. As mentioned above, with virtual subsystems the model hierarchy is flattened and execution order is determined solely by a built-in data dependency sort. Conversely, with atomic subsystems the algorithm designer has much more control. Though the ability to explicitly define block execution order does not currently exist, there are practices, and more importantly model architectural decisions that can be used to specify this behavior. In fact, it would seem highly desirable if the model architecture enforces some rules about how the signal flow should be modeled, such that the block execution order is intuitive and completely deterministic based on the model itself. This would prevent the need for the capability to explicitly set execution order, a capability that is only necessary if the algorithm is modeled in a nondeterministic manner. Whether it is possible to define the execution order via the model is highly dependent on the algorithm being modeled, but there are some block constructs and stylistic guidelines that make this approach easier.

Before attempting to control execution order, however, it is helpful to understand how Simulink makes this determination. The concept of signal flow is how the data represented by signal lines in Simulink moves from block to block. In other words, it is an expression of the external data dependencies of each block. Figure 1 shows an extremely simple model, consisting of only two virtual subsystems in a feedback loop.

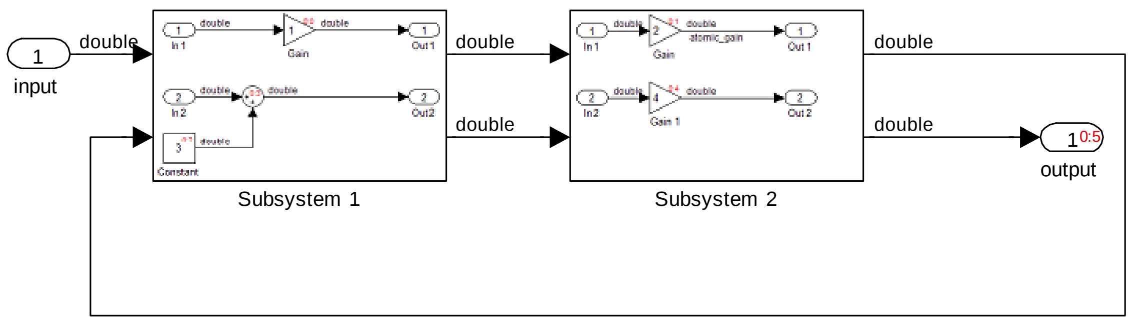

Figure 1: Simple discrete model

Notice the sorted order displayed in the upper-right corner of the blocks. In this case the execution order is:

1. Subsystem 1: Gain
2. Subsystem 2: Gain
3. Subsystem 1: Constant
4. Subsystem 1: Sum
5. Subsystem 2: Gain1
6. output

This execution order may seem surprising to some algorithm designers, who expect Simulink to do better job of "understanding" the model and its intent. In fact, some users may go so far as to say that this execution order is simply wrong. However, recall the discussion of virtual subsystems above, where it was stated that they are solely for graphical organization and have no functional meaning. This is the perfect example of that fact. Further insight can be gained by looking at this exact same algorithm with only some graphical changes that make the signal flow much more apparent. Figure 2 shows this case.

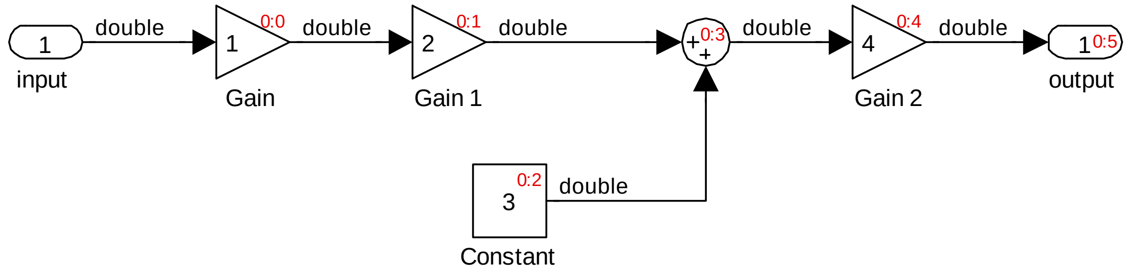

Figure 2: Sample discrete model, graphical changes only

Comparing Figures 1 and 2 shows that indeed the algorithm is exactly the same, only the virtual subsystem boundaries have been removed. Furthermore, the sorted order is exactly the same. However, if Figure 1 had not been seen, one might indeed think that the sorted order shown for the model in Figure 2 is perfectly proper. This example illustrates the single most important suggestion for a successful model architecture: make the model fully self-contained, deterministic, and intuitive. However, this suggestion first requires a fundamental understanding about Simulink, one shown by the preceding example. In Simulink the graphical layout of the blocks has no affect on the algorithm being modeled, particularly with respect to execution order. More simply, in Simulink, block position has nothing to do with execution order! (Note that this is different than with Stateflow.) This fact may seem counterintuitive because many engineers, instinctively and regardless of experience, want relative block positions in the model to be a meaningful representation of relative execution order. This example demonstrates why the most important suggestion for a successful model architecture is to make the model fully self-contained and intuitive. In other words, if it is commonly expected that the relative positions of blocks within a model should depict the execution order, then the signal flow of the algorithm should be modeled such that this is the case. Giving the subsystem boundaries shown in Figure 1 some functional meaning is accomplished in Figure 3 using atomic subsystems. Because the atomic subsystems act as a functional group, the algorithmic meaning of the model in Figure 3 is completely different than in Figure 1 despite the fact that no signals or blocks other than the subsystems were altered (a unit delay is required on the feedback path to avoid an algebraic loop). Though their functional meanings are different, in both Figure 2 and Figure 3 the sorted order is completely deterministic due to proper modeling of the signal flow. Furthermore, the sorted order conforms to the expectation derived from the relative block positions.

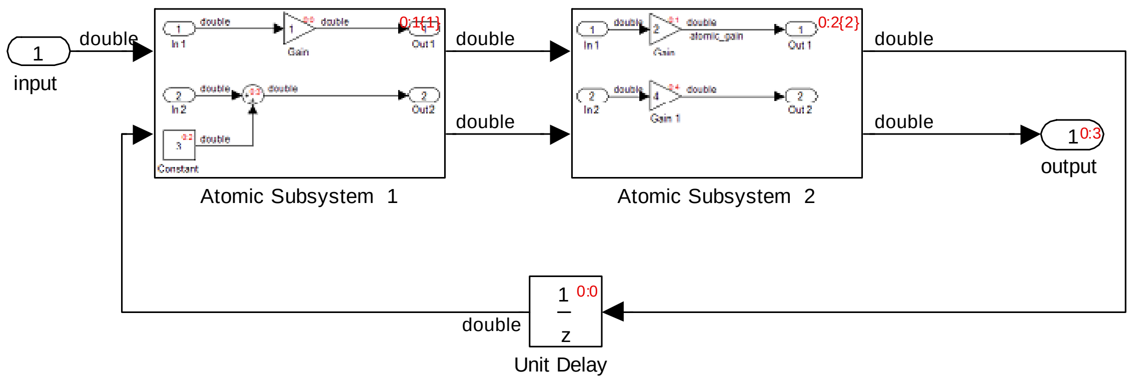

Figure 3: Sample discrete model, functional boundaries

Deterministically modeling the signal flow in the model environment is especially pertinent when working in a high integrity environment, where determinism is essential. By not clearly modeling the signal flow, as in Figure 1, the algorithm designer forgoes any understanding or control over the relative execution order of the algorithm. Furthermore there is some ambiguity about the intended algorithm. Consider a more traditional software development process, where a software engineer would take the block diagram in Figure 1 and write code by hand to implement that algorithm. Depending on the interpretation, the code could implement the algorithm in Figure 2 or the algorithm in Figure 3! Luckily, this issue is avoided by automatically generating code from a model. When automatically generating code from the model in Figure 1, the only way to have the same functionality is to generate a single function for the entire model. Obviously for large models in a high integrity environment the verification aspect of such a process is prohibitive. Avoiding generation of a single function by attempting to use the subsystem boundaries in Figure 1 as code generation boundaries would prove disastrous. If code were generated individually from the subsystems in Figure 1 and then integrated in a software environment, the resultant algorithm in the code would behave exactly as the algorithm modeled in Figure 3. This is clearly different than the algorithm modeled in Figure 1. This illustrates the importance of consistency between the model architecture and the software architecture. The above example also illustrates that the most effective way to control execution order in a model and provide the same functionality in the generated code is by deterministically modeling the signal flow.

The ability to set block priority is another capability in Simulink to help specify execution order. We note that setting block priority is not recommended as a substitute to modeling the signal flow, but as a best practice in conjunction with signal flow modeling. With atomic subsystems, the algorithm designer has the capability to set an execution priority for that subsystem. This priority is essentially a tie-breaker for the built-in data-dependency sort algorithm in Simulink. Simulink will still decide execution order based on the signal flow expressed in the model. However, in the event that two atomic subsystem's relative execution order cannot be fully determined by the signal flow, Simulink will refer to the aforementioned priority to decide which atomic subsystem to execute first. This practice, therefore, allows the designer some control over the block execution order, subject to the constraints of the signal flow expressed in the model. Thus, the ideal model architecture should enforce some rules about how the signal flow should be modeled such that the block execution order is completely deterministic based on the model itself.

CONCLUSION

This paper covered topics related to componentization and modularization of a model, and how these topics influence issues like execution order. Clearly, there is a great deal to consider within the large topic of model architecture. However, the practice of using a combination of model referencing and atomic subsystems in a flexible but hierarchical manner provides a significant amount of capability to address these issues as well as some configuration management and simulation performance issues. Furthermore, the fundamental concept for a successful model architecture was introduced: The model must be a self-contained, fully-specified, deterministic, hierarchical description of the algorithm.