A Proposal for a Model Compiler

William P. Milam[1]
and
Alongkrit Chutinan[2]

Powertrain Control Systems Department
Ford Research Laboratory

1 Introduction and Motivation

With the widespread adoption of modeling as a tool for analyzing and designing systems an issue has come to light. The issue is how to reliably build and maintain models from components. Many people within the Ford community are generating models to help solve their local problems, however we are unable to effectively leverage this large distributed set of models due to a lack of tools and infrastructure. Of course the fact that there is no uniform modeling style, even when using the same tools and language, adds to the problem. So while we have a large base of models used to solve everyday engineering problems, we lack the ability to use these models beyond the initial application.

In many ways this is equivalent to the dilemma faced by the software world. Programs, and parts of programs, were written to solve immediate problems. When a similar problem surfaced yet another program would be written. The two main reasons were a lack of knowledge that previous solutions existed and that there was no direct way to reuse the relevant parts of the program in the new solution. In the Eighties the use of High Level Languages(HLL) was advocated to allow programs to be written in a more abstract form. These abstract programs could be easily recompiled for various platforms thus enabling their reuse. Unfortunately the need to produce new software products from existing software increased beyond the point that HLL's could address the issue. Now people were more interested in reusing portions of programs. From the need to address this problem was born the object-oriented approach to software development.

What we would like to propose is a solution to the modeling dilemma based on a combination of High Level Language and object oriented methodologies. This paper will give a brief overview of the methodologies and then outline how the methodologies can be applied to the problem at hand.

2 Object-Oriented Methodology

In the object-oriented paradigm the programmer was encouraged to try and think of items to be processed as objects. For example if we were writing a program to process vectors, and there is no native support for a vector in the language. Can we create a data type that represents a vector?

[1] wmilam@ford.com
[2] achutina@ford.com

The simple answer is yes. You can do that without object oriented support. However if we want to create an object vector and 'build' in support for operations on a vector type the object oriented paradigm is supremely suited.

For the vector object we will define certain properties of vectors. These would be size, type, name and values. The number of values in a vector would be captured by size. The idea of a type of vector might be whether a vector was comprised of integer or floating-point values. Let's say we wanted to add two vectors. If we take a page from group theory and think of vectors as our members and the operation of vector addition as the operation then we could add vectors with ease. By casting vectors as a numeric type with similar operations as we would expect for integer scalars, we create a reusable bit of software that can be reused without the user having to be intimately familiar with how it performs vector operations. Similarly we could also have overloaded operations for adding a scalar to a vector. So instead of having to dream up a routine to add a scalar to a vector, which might look like:

New_Vector = add_scalar(vector, scalar);

We might just use:

New_Vector = vector + scalar;

New_Vector = vector1 + vector2;

The last two are examples of adding a scalar value to a vector and adding two vectors together. Both are typical operations and are far easier for the user to apply. If the object definitions were shared with a subsequent user they too would be able to apply the vector operations in a simple and intuitive manner. Certainly the ability to reuse the vector operations saves a great deal of time and effort.

We can treat Simulink[3] models in a similar manner. Each block could be thought of as an object. Not every block would be a unique object however. For example the sum and product blocks are very similar. In fact the only real difference is in the operation performed. So we could think of them as one object with an attribute that indicates the operation to be performed on the inputs. This then gives us one object to represent either block. Another simple example is the inport and outport blocks. Each block has a similar function, the key difference being direction of data flow. It is either into or out of the subsystem.

To carry it further all the Simulink blocks could be represented by an object a key attribute of which is block type. As each block corresponds to an object the object could also have architectural information, such as the subsystem the block is a member of, what are the peer blocks at this context level and others.

This does not preclude the idea that there may be Simulink subsystems that are indeed treated as a traditional object. For example if we have a subsystem that represents a C cell battery. We might well want to model some systems that use more than one battery. Also we may wish to

[3] Simulink and MATLAB are registed trademarks and Stateflow is a trademark of The Mathworks, Inc.

wire the batteries together in a series or parallel configuration. In this case we can imagine that it would be useful to have an object class called C-cell. We will then instantiate C-cell objects and also utilize operations, such as series, to indicate how they are to be configured.

3 Describing a Model

In order to solve the issue of building a model from components at least two things must happen:

1) A definition of a model and model components must be agreed.
2) A mechanism for describing a particular model instantiation, built from components, must be agreed.

So first we shall define a model. This definition is only for the purposes of discussion in the context of this document. As we are only working in the automotive Computer Aided Control System Design (CACSD)[i] context we can keep ourselves restricted to models implemented in the CACSD environment. In that case we can define a model to be any CACSD conformal model that has no explicit external inputs or outputs. So a model that looks like figure 1, which has no input or output ports, would be a trivial example of a model. The actual model style, labeling of signals, use of control and data flow signals, etc is determined by a style guide[ii]. The Mathworks, authors of Matlab and Simulink, have proposed a common style guide for use by the automotive industry. This guide was prepared with involvement by several automotive OEM's.

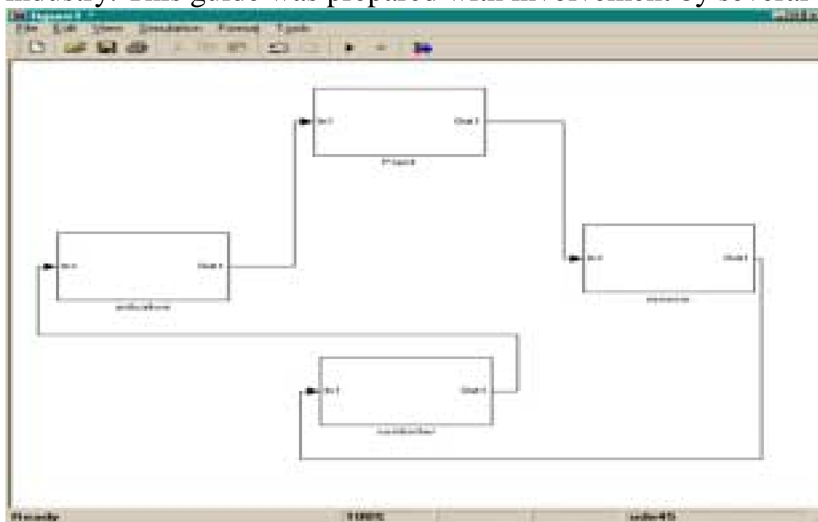

Figure 1

Well that seems simple enough, except that the model has properties. One of them is what components does it contain. This leads to a definition of a component. A component is a CACSD sub-system, or a base Simulink block. If it is a subsystem then it has inputs and /or outputs. To illustrate see the visual examples in Figure 2.

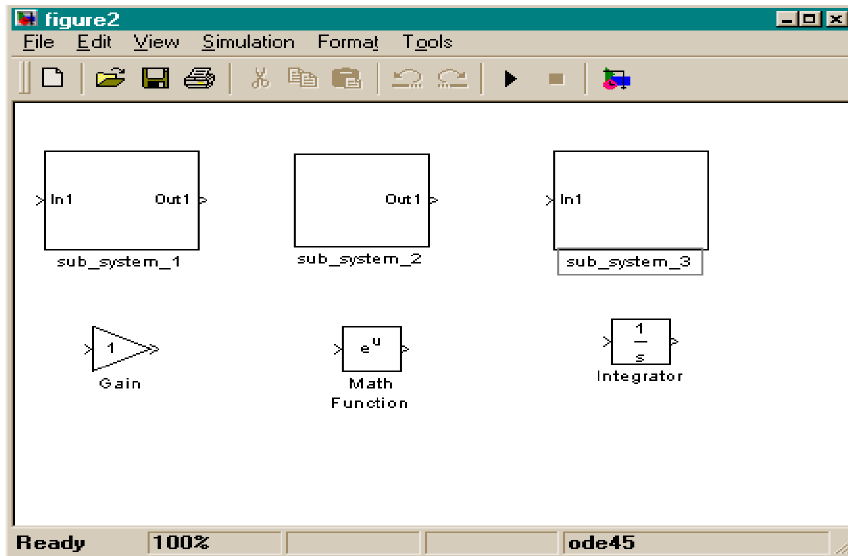

Figure 2

There are six components shown in Figure 2. The top row of blocks are examples of the sub-system type component. Note that each example has at least one input or output. In addition each sub-system component may also have a trigger port. This allows the sub-system to be run in a deterministic manner. The bottom row of blocks are standard Simulink blocks. For purposes of this paper I will refer to them as terminating blocks. By that I mean that there is no further recursion possible past a terminating block.

4 Formal Language to Describe Model Architecture

This leads to a recursive definition of a CACSD model, which contains components that ultimately become terminating, or base, blocks. With a recursive definition we can define a formal language to describe the relationships between model components.

One of the base tenets of language processing is the concept of a grammar. It is similar to the common idea of grammar as applied to English. You have an *alphabet*, but it would be more akin to a vocabulary. A vocabulary, for our purposes, can be defined as a finite set of symbols, or words. These symbols can be arranged in strings, sort of like a sentence. So let's define two symbols, V and V*. These represent our Vocabulary and the set of all strings that can be formed from V respectively.

To put this in more concrete terms lets say that we define V to be the following:

V = {Ken, Bill, or, jumps, quickly}

So now V* would have strings like:

1) Ken jumps quickly or Bill jumps quickly

2) Ken jumps quickly
3) Quickly jumps Ken
4) Bill quickly jumps or

Strings 1 and 2 make some sense. String 3 makes an interesting name perhaps, but string 4 is not an acceptable sentence. How do we determine that? Well we do that by applying rules, or *productions*, to the vocabulary V to select the language L that is a subset of V*.

For a more detailed overview of the whole idea of defining grammars for a formal language I suggest you seek out a computer science text[iii]. For our purposes we can declare that for a string to be considered valid for our example language it must follow the following rules:

1) A valid ***sentence*** has a ***noun*** followed by a ***verb phrase***.
2) A ***noun*** is Ken.
3) A ***noun*** is Bill
4) A ***verb phrase*** is made up of a ***verb*** followed by an ***adverb***.
5) A ***verb*** is jumps
6) An ***adverb*** is quickly.

So if we apply the rules above only string 2 is valid. String 1 uses **or** which is not covered by any rule and should not be accepted. If we were to add rules to define **or** as a conjunction and also that a valid sentence could be a sentence followed by a conjunction followed by a sentence, then string 1 would also be valid.

We can define parsers to process these rules over defined vocabularies. While this may seem clear to you for application to grammar checking and compiler building, the real question is how can we adopt these concepts to the problem of defining a model and how it is architected.

5 Models as Collections of Objects

Earlier we defined model components. These components can be thought of as being equivalent to symbols. This leads to a definition of a vocabulary that consists of CACSD compliant components which can be arranged in such a way so as to create a model. Now if each component is thought of as an object, it will have certain properties that can be operated on to produce a fully wired model with all inputs and outputs matched.

Each object corresponds to a particular model component. Some of the key object properties, which can be derived from a CACSD compliant model, are:

1) Pointer to parent
2) Pointer to peer(s)
3) Pointer to child
4) Sample Time
5) Set of inputs

a. Data dictionary definition for each input
    i. Name
    ii. Type

6) Set of outputs
    a. Data dictionary definition for each output
        i. Name
        ii. Type
        iii. Source component

7) Type of block
    a. Sub-system
    b. S-function
    c. Stateflow
    d. Etc.

8) Set of workspace parameters (typically calibration)
    a. Data dictionary definition for each parameter
        i. Name
        ii. Type

So let us take a component and see how the object data maps onto it. If we have a component, which could be called Torque_Truncation_Spark but we will just use X, we first have an object that corresponds to X. If we have this component in it's own context then the parent and peer fields are null. The child pointer will point to some internal component. It does not matter which one so long as it is in the next hierarchical level. So look at figure 3.

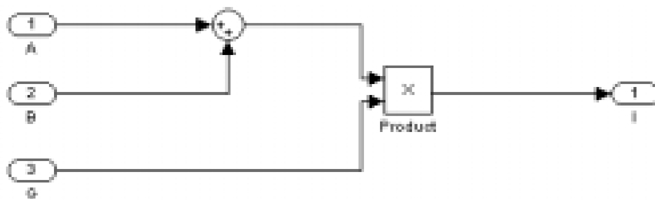

Figure 3

If we set the child pointer of X to point to the first inport block then we have started the chain that will point to all the other children of X. This would be done by creating other objects which represent the summation, product, inport and outport blocks. What we would have then is a tree structure of objects which are related. It would look something like the following.

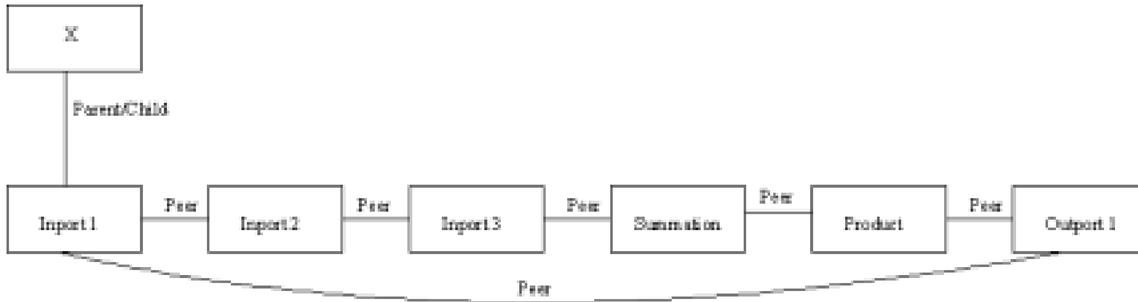

Figure 4

Each object is represented by an identical block as they are indeed identical for our purposes. The contents of the object reflect the block it represents as instantiated. In parsing they talk about tokens, and these are our tokens. It is by processing the tokens that we create the output model that we wish to construct. It is not realistic to consider that all models will be built in this manner from base Simulink blocks. However when you have a large library of existing models that represent various functions, you would likely use this to generate larger models that contain the pre-existing components. You may also include some Simulink blocks as part of this construction process.

6 Generating Larger Models from Bookshelved Components

Take the example where we have several model components, how would we combine them to create another component? The obvious solution would seem to be to combine the model component as object concept with a formal language that describes the architecture of the model created by the components. How might this work? Let's create a simple example based on three simple components. Each has some number of inputs and at least one output. We can see them in figure 5.

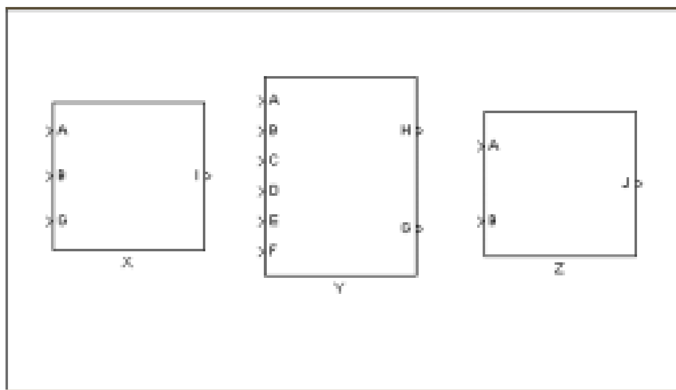

Figure 5

Here we see components X, Y, and Z. Each has inputs and outputs. Each input and output is assumed to be a scalar type. In addition each input and output is named with a signal name. For our example these are just simple single character names, however in a real model they might be RPM, Coolant Temperature or Throttle Position.

Now if we define a component to have certain characteristics, the set of inputs and the set of outputs, we can walk through the following scenario.

Let's for the moment accept that the statement:

Q { X,Y,Z}

Means that we are creating a component Q with members X, Y and Z. That would look a lot like Figure 5 if we think of the figure as representing Q. So what is the set of all inputs to Q? We can find that by taking the sets of inputs from the three components and looking at the union.
So we have:

$X.in = \{A,B,G\}$
$Y.in = \{A,B,C,D,E,F\}$
$Z.in = \{A,B\}$

$$Q.in = X.in \bigcup Y.in \bigcup Z.in = \{A,B,C,D,E,F,G\}$$

So Q.in represents the set of all inputs to Q. We can perform the same action for determining the set of outputs for Q. If we look at the diagram we see that G is actually an output from Y. How can we recognize that? By finding the intersection of Q.in and Q.out.

$$Q.out = X.out \bigcup Y.out \bigcup Z.out = \{G,H,I,J\}$$
$$Q.inner = Q.out \bigcap Q.in = \{G\}$$

So Q.inner represents the set of signals that originate and terminate inside Q. Another reasonable action might be to remove G as an element of Q.in and Q.out. If G originates within Q, it certainly should not also be a member of Q.in. It may, or may not, also be a member of Q.out. We could remove it from that set as a default. Of course it may be possible that G is used by another component, which is a peer of Q. In that case we would need the ability to search a model to find instances of G that are a match in both name and type to the G that is needed.

As this demonstrates by applying simple operations on component attributes we can 'wire' together the components to form either larger groups of components or a full model. As these functions are recursive it is now possible to define a language to support this task of combining model components according to some rules. The use of rules implies that errors can be identified and flagged. This would be similar to the function of a HLL compiler which does matching between prototype function declarations and instantiations of the function call.

If we take the case where we have a predefined architecture the error capability might become clearer. Let's add some syntax to our example above. Where we had:

Q{X,Y,Z}

Now we have:

Q(A,B,C,D,F;G,H,I)
{X,Y,Z}

So Q will still be a sub-system made up of X, Y and Z. The parenthesis tells us what the defined inputs and outputs are for Q. So we have Q.in predefined to be {A,B,C,D,F} and Q.out is defined to be {G,H,I}. If all is well the Q.in should match the Q.in as built from X,Y and Z. We can test by checking if the union of the two input sets is equal to the intersection of those sets. The simplest test is comparing the cardinality of the intersection versus the cardinality of the union. If those are equal then there are no errors. However once we determine that there are errors what kind of errors can we find?

By using set difference we can determine which signals were needed but not defined. For example if we think of there being two Q.in sets, with one being Q.in_defined and the other Q.in_found. Then we take two differences. One is the difference from Q.in_found – Q.in_defined. The resulting set would be all the signals found to be needed as inputs, but which were not predefined. In our example that would be input E. This could be an error on the part of the person who defined the input signal set, or it could be that a component is incorrect. Either way the system is able to identify such things. This could be very useful in testing if new components conform to a predefined architecture.

Another area of analysis that becomes easier to perform is in checking for scheduling issues. The simplest example might be where we take the following to be true:

1) Q.sample_time = 1 msec
2) X.sample_time = 10ms
3) Y.sample_time = 1 ms
4) Z.sample_time = 5ms

Now we could do a simple test to verify that there is no inherent conflict between Q and the children of Q. The obvious test would seem to be performing a modulus operation of each child sample time versus the parent sample time. If the result for all three is zero, then there is no scheduling conflict. This is because each child sample time is either equal to, or a multiple of, the parent sample time.

With that determined it would be possible to generate a Stateflow scheduler which would execute the child models according to the relationship between the parent and child sample times. Also it would be possible to sequence the models to allow for data dependency. For

example the G signal from Y is used by X. So it would seem reasonable to execute Y before X so that the G signal is up to date with respect to the A and B signals used by both X and Y.

If we take another view of the same information we can also determine if there is any relationship between G and A,B. To do this we can apply the same analysis to component Y we used in building component Q. This of course assumes that component Y is another CACSD compliant model. If it were a s-function based on implementation code the component is treated as a black box and the exact relationship between input and output is not available. So while we could use either implementation code or a model, there are differences in the depth of analysis available.

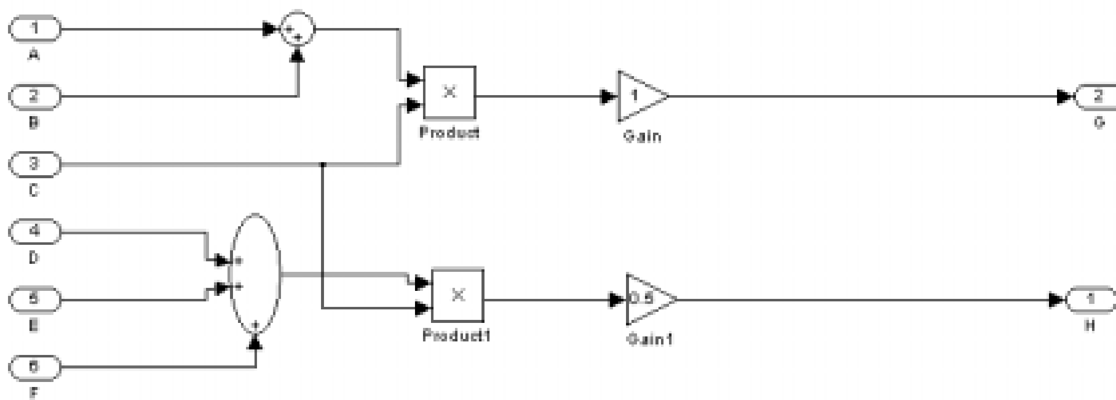

Figure 6 (Inside sub-system Y)

As we look at the contents of component Y it is clear that G = C(A+B). So indeed G is dependent on A and B. Each line in the model represents a unique signal. For example the line connecting the output of the block 'Gain' to outport 2 represents a value that is different from the line connecting the output of 'Product' to the input of 'Gain'. Each line connecting any blocks can be thought of as an edge of a directed graph. With each block serving as a vertex, a graph of the data flow into G would look like Figure 7.

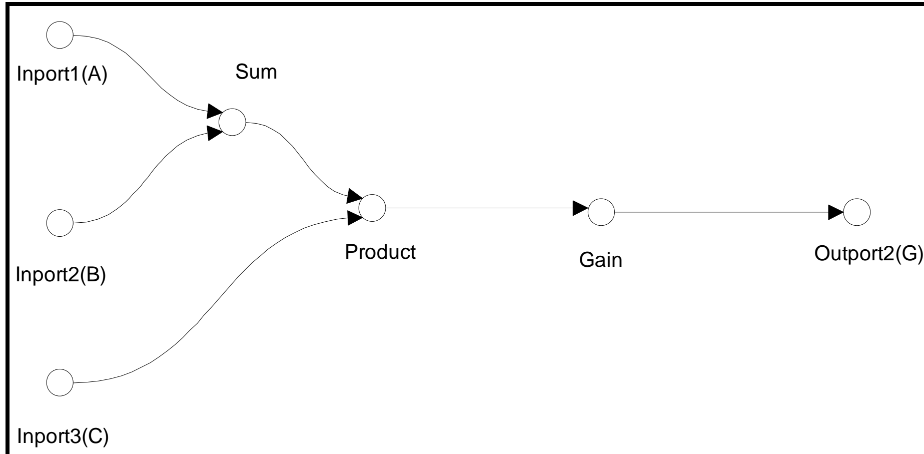

Figure 7 (Directed Graph of the Calculation of G)

This type of representation would have to be built as each sub-system was 'parsed'. This leads to the conclusion that we will need a two part process. One part will parse an architecture language that describes relationships between components. The other will parse the components to create mappings of relationships such as in Figure 7. Then the architecture language will be a control language to the output generator which will take the parsed components and create a new component as directed by the architectural description. We can also then perform analysis on the components to determine if the components being combined are compatible. The analysis outlined in the previous text would be among the many possible. This is not exactly the kind of function one would typically think of for a compiler.

7 Conclusion

Modeling tools today have proven their value in allowing engineers to model and analyze the behavior of complex systems. However the ability of the company to leverage this knowledge through the reuse of these models is severely limited. Even with the adoption of modeling standards, such as the Mathworks Automotive Advisory Board Guidelines, the ability to recombine these models into larger and more complex models is fraught with peril. The process of composing larger sub-systems is entirely manual at this time, which puts a significant roadblock in the path of common use, and reuse, of these models and the significant investment they represent.

To help illustrate this might be accomplished, we will include a sample of the example problem formulated in section six. This is only a prototype showpiece to demonstrate what could be done in the Matlab environment as it exists today. If you do not have the example files, please contact the authors. We will also follow up this paper with further work outlining the specific types of analysis we would like to see in a tool.

We hope that the ideas and concepts presented here will serve as a basis for a new line of development for tools to help engineers make better use of existing models by freeing them from the tedious manual work while adding significant value in analysis.

[i] Smith, Patel, Sun, Ramanan, Donald, Toeppe, Ranville, Bostic, Butts, "CACSD in Production Development: An Engine Control Case Study," Proceedings of the Global Powertrain Congress 2000, Detroit, MI, June, 2000.

[ii] Mathworks Automotive Advisory Board, “MAAB Style Guide”, available first quarter of 2001 from The Mathworks.

[iii] Thomas A. Sudkamp, “Languages and Machines, An Introduction to the Theory of Computer Science” Second Edition, Addison Wesley.