

Object-oriented approach to the development of level-2 M-file S-function

Gianmarco Romano

(Assistant Professor, Dipartimento di Ingegneria dell'Informazione, Seconda Università di Napoli, Aversa (CE), Italy)

Abstract

We present an object-oriented approach to the development of custom blocks in Simulink. Matlab provides an object-oriented (OO) programming model, as the M language provides classes and other OO concepts such as inheritance and interfaces, but Simulink M-file S-function cannot take advantage of such features. Custom (M-file) blocks cannot be implemented as a class, but single (callbacks) methods must be implemented, and internal state, or any data that must be passed between calls backs in a S-function, can only be represented by simple data type: for example, DWork vectors in Level-2 M-file S-function cannot be of class data type and no analogous of PWork, available for C++ S-functions, exists. As the availability of OO Matlab programs increases, a way to re-use Matlab classes in S-function is needed in order to avoid a complete rewrite of algorithms as simple functions. We show how to overcome such limitations by providing an example of synchronous CDMA Simulink model that use MATLAB classes to implement custom blocks in Simulink. Such an approach can ease and accelerate the development of custom blocks especially for programmers accustomed to the modularity and abstraction of OO languages and can provide scaling and productivity gains as those achieved by OO languages.

Keywords

Object-oriented, custom blocks, classes, CDMA

1. Introduction

Matlab has added new object-oriented features to its interpreted language M in order to gain advantage from the object-oriented approach. Object-oriented software design methods, such as design patterns [Gamma *et al.*, 1995], can ease the development of MATLAB programs and toolboxes, and can help to manage the increasing complexity of code. Object-oriented extensions are relatively recent and offer only some of the features of other object-oriented languages, such as C++ and Java, though more features could be added in the future. Nevertheless it can help improve the productivity of the development cycle and it is expected an even more widespread use of classes in MATLAB in the future.

We use MATLAB object-oriented features to write code for simulation of communication systems, as for example CDMA systems, and therefore we have developed a set of classes that represents various components and algorithms. Because a communication system model can usually be represented as a block scheme, we implemented dataflow design patterns. Simulink is the tool of choice for simulation of systems that can be modelled as block scheme. Therefore is well suited for simulation of communication systems and also for development of implementation code for embedded systems, through automatic code generation. Many existing blocks are available as built-in blocks and as libraries, called blockset. For communication systems many blocks are provided in the communication blockset and can

be used to model a variety of communication systems. Nevertheless there is a need to develop custom blocks that implements specific and new algorithms. Simulink offers the possibility of developing custom blocks. There exists a variety of methods to develop custom blocks, that we will discuss in Section 2, however our focus in this paper is on how to develop custom blocks by using object-oriented extensions of the M language. We faced this problem in the transition from Matlab object-oriented programs written to simulate communication systems to Simulink models, where the problem of the re-use of our code arises. Specifically we consider the following problem: assume there exists some legacy code developed by using the object-oriented extensions of the M language, i.e. there exists a set of classes, how can you use that code, unmodified, in a Level 2 M-file S-function (custom block implementation) in Simulink? The problem arises from the fact that custom block can be developed in M language, but no explicit support for classes in S-function is provided. The solution to this problem could accelerate the transition towards the adoption of Simulink as simulation tool for experienced and advanced MATLAB users. Many prefers to keep on using Matlab because they do not want to, or simply cannot, rewrite code. We will show how to easily incorporate classes in S-function.

This paper describes our experience and solutions and our results, though referred to a specific Simulink model, can be extended and applied in more general contexts.

It is worth noting that we do not consider the problem of providing object-oriented approach to the graphical modelling, i.e. apply object-oriented traditional concepts as classes, inheritance, interfaces and polymorphism to the graph, as proposed for example in [Lee2004]. We restrict our attention at the implementation of custom blocks and even in this simpler scenario we show that valuable benefits can be gained.

The paper is organized as follows. In section 2 we review some of the methods to develop custom blocks in Simulink, in Section 3 we illustrate our object-oriented approach to the development of custom blocks, and in Section 4 we draw the conclusions.

2. Custom blocks in Simulink

In Simulink a block is any object that has at least one input and/or one output and that can process data. Blocks can also have a state, which constitutes a memory for the block itself. Many blocks are provided by Simulink as blockset, i.e. library of blocks, and can be used out of box for most system models. For more advanced systems existing blocks might not be sufficient and therefore there is the need for users to develop new blocks.

Simulink offers as one of its main features the possibility of developing custom block as a way to extend the basic functionality provided by built-in blocks. There exists several methods to create custom blocks. They can be implemented as subsystems, i.e. as models built from existing blocks, with several inputs and outputs. Alternatively, they can also be implemented in different programming languages as C/C++, Fortran, ADA, and M: whatever the language chosen, one needs to write an S-function. This approach is preferred when there is some existing code to be re-used, such as legacy code or some library, or the algorithm on which the block is based is better and easier implemented in a traditional programming language, or it simply cannot be implemented using existing Simulink blocks; another motivation to use a programming language is to achieve better performances in terms of speed of execution. When implementing a custom block by a programming language, one can take advantage from the features of the language itself. So, for example, if one uses C++ then one can benefit from all the object-oriented features to manage the complexity of the code itself and can use recent design methods to develop pattern architectures. By choosing the M language, one can take advantages of the interpreted nature of the language, development tools such as the debugger, and the availability of toolboxes, typically at the cost of worse performances.

Depending of the complexity of operations to be performed by the custom block, several options are available to develop a custom block in M. The easiest is to define the block in terms of an M-function with one input and one output and no states. Simulink provides simple blocks, `Fcn`

or `MATLAB_Fcn`, that just require to specify a MATLAB expression or function, respectively. This method is well suited when you have an existing MATLAB function to be part of the model or when it is easier to write M code rather than use existing blocks. Both methods are possible when no states, either continuous or discrete, need to be defined. A similar method is aimed at embedded systems, by using a block called `Embedded MATLAB`. Again a function defines the operation of the block but only a subset of M language and toolboxes are available. Such a restriction is due to the fact that this block automatically generates code for embedded systems.

Greater flexibility is provided by Level 2 M-file S-functions. This method provides access to the S-function API, which defines at deeper level the behaviour of any Simulink block. Customization mainly consists of providing implementation of a set of functions, defined by the S-function API, that are called during the cycle of life of each block by the Simulink engine. Some of them are called during the initialization stage, other during the simulation loop. At the end of each simulation, some other functions are called to clean up the system (for a list of S-function API refer to the Simulink documentation). At the initialization stage block features as the number of input and output ports and their data types, must be defined.

In M to have access to the S-function API, callback functions need to be registered and implemented. In the simplest case, just one callback needs to be registered: the method called to generate the output. More generally, when, for example, also a state is defined, some information need to be stored and be available across callbacks and/or calls of the same callback.

In Level 2 M-file S-functions, all information about a block, as for example run-time data and parameters, is stored in a built-in *data class object* (`Simulink.MSFcnRunTimeBlock`), that is instantiated by the Simulink engine. `Simulink.MSFcnRunTimeBlock` run-time object is also used to make available to all callbacks any run-time information, since this object is passed as argument to each callback. In Level 2 M-file S-function no global variable can be defined and therefore any custom data that has to be accessed and visible in more than one

callback or across multiple calls of the same function, must be stored in any of the field of `Simulink.MSFcnRunTimeBlock`.

One example of data that need to be stored across multiple calls is the state of the block. For such a purpose a specific field of `Simulink.MSFcnRunTimeBlock` run-time object exists: `DWork Vectors`, which are array that can take on values of only some data types, Any other data type, as for example MATLAB classes, cannot be used in such a field.

3. Custom blocks as classes

Conceptually a block can be thought as an object in the sense of object-oriented programming, because its behaviour and interactions with other blocks are well described by an interface and its fields. Therefore object-oriented implementation of blocks seems natural as new custom blocks have to be developed. Simulink, however, lacks object-oriented support at Level-2 M-file S-function level, so some workaround needs to be found to make object-oriented programming possible.

In order to illustrate our approach to object-oriented developing of custom block we refer to a communication system model (synchronous CDMA system) where a custom block implements a multiuser receiver (Fig. 1). Its interface is very simple as the main operation is to decode received data and one method can implement specific algorithm and provide the decoded data. Differences among several receivers are taken into account in the implementation of the decode method and no change in the class interface is needed. Therefore the block that implements the receiver has one input and one output and no internal states are defined. Our focus on a simple block is motivated by the fact that we want to illustrate the method of object-oriented development and we do not want to enter the details of developing a full featured custom block.

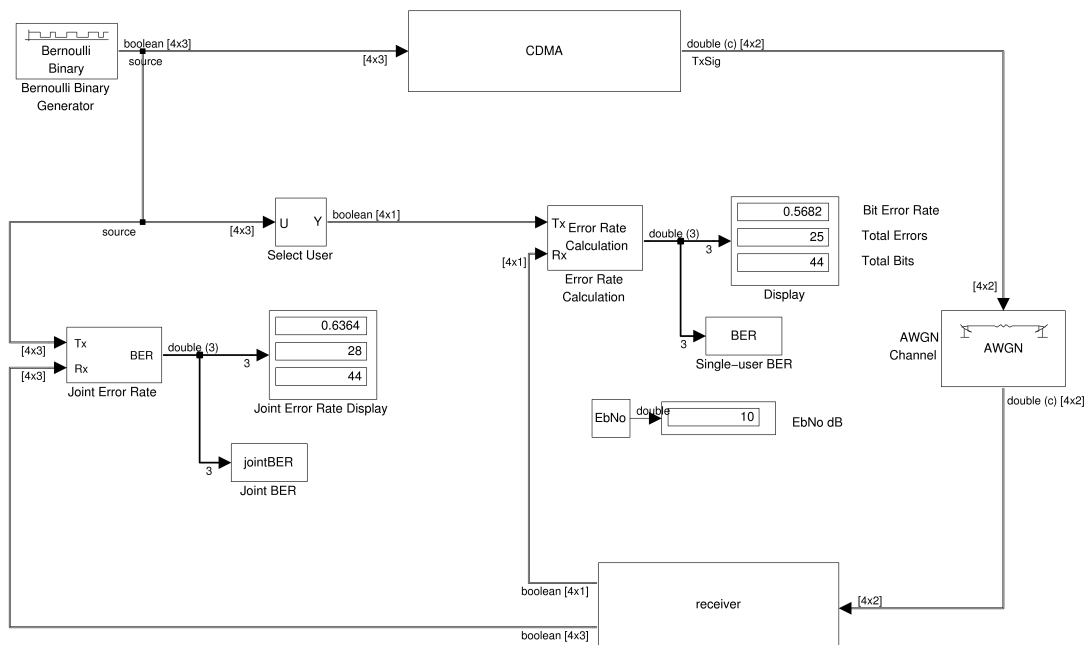


Figure.1 Simulink model for synchronous CDMA systems

The basic idea of our method is to call class methods from within callbacks, resulting in a S-function that becomes just a wrapper to the interface of the implementation class. For example, in the callback that generates the output data (we call it `Ooutputs`), a call to the `decode` method of the class that implements the decoder is what we need. To make this call, however, we need a handle to the class to be available in the callback.

An efficient approach is to initialize the class when the block is initialized, and store the implementation class somewhere that is visible in each callback, such that the appropriate method of the same instance of the implementation class can be called. Unfortunately no global variables, that could store the implementation class, can be defined in Level 2 M-file S-functions, and therefore some other way to store the class has to be devised.

In C++ S-function Work Vectors offer a way to store data along all the life of the block. They can therefore be used to pass data between functions. For example, in C++ pointers to

classes can be stored in a pointer array (`Pwork`) that is visible and accessible in any of the static functions whose implementations define the behaviour of the block. A number of helper function support the use of the `Pwork` as a way to pass a reference to any function. This way a public fields and methods of the implementation class can be accessed by the S-function. Work Vectors are available also in Level 2 M-file S-function, but unfortunately only `Dwork` vectors are available. `Dwork` supports only a limited set of data type, and class objects are not included. Therefore cannot be used to solve our problem. We need to find some other mechanism to store class objects.

Work Vectors, as all data and parameters related to a block, are stored in `Simulink.MSFcnRuntimeBlock` run-time object. Since this object is passed to each callback, we need to find some other field that can store classes. There exists a field that can help in our case: `UserData` field can store any object of any data type, including classes.

UserData field can be set at the initialization stage of the block. For such purpose instantiation code can be defined in the callback function `InitFcn`, in the block properties as shown in Fig. 2. `MLdetector` is the implementation class for the decoder and just needs a class `c`, defined in the workspace, for its instantiation.

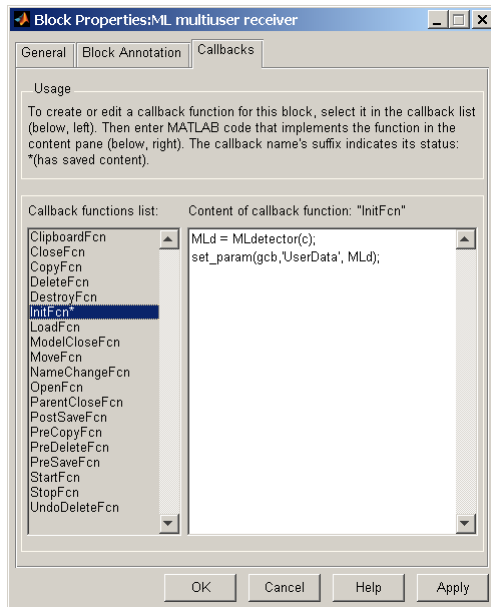


Figure 2. Initialization of implementation class

Now the implementation class can be retrieved in the Outputs callback by getting `UserData` field of `Simulink.MSFcnRunTimeBlock` run-time object, as shown in the following code

```
% get decoder implementation class
d = get_param(block.BlockHandle, 'UserData');
```

Then a call to the appropriate method `decode` of the implementation class can be made

```
% call implementation class method decode for
each sample of the frame input
for k=1:block.InputPort(1).Dimensions(1)
    block.OutputPort(1).Data(k, :) =
    decode(d, block.InputPort(1).Data(k, :))';
end
```

The for loop is needed in this case because the input signal is a frame.

Our solution is, therefore, as follows: instantiate the implementation class and store the class in

the `UserData` field of the `Simulink.MSFcnRunTimeBlock` run-time object at the initialization of the block (`InitFcn`); in each callback body retrieve the the implementation class and call the appropriate method.

As long as the interface of the implementation class remains the same, a change in the implementation of the custom block can be made just by changing the class to be instantiated. For example in `InitFcn` instead of

```
MLd = MLdetector(c);
```

a different implementation class for a different receiver, as for example linear multiuser receiver, can be instantiated as follows

```
MLd = Ldetector(c);
```

No change in Level-2 M-file S-function is required. Therefore one of the benefit of this solution is that one can choose the implementation class as a parameter of the model or simply the implementation class can be chosen from a mask, or the implementation class can be set in a script file that runs the simulation. For example, in wireless communication system simulations one needs to compare the performances of different receivers and a simulation script can generate results for different receivers. We remark that as all decoder classes have the same interface no change in S-function is required.

As classes are written in M, debugging can be performed by using the integrated debugger in MATLAB. More generally all the tools available for development of M-files are clearly available also for classes.

Our object-oriented approach to the development of custom blocks can also take advantage from the fact that some or part of the method of a class can be implemented in C/C++. This allows a mixed developing process that can represent an interesting compromise between performances, in terms of speed of execution, and complexity, and then time to develop and maintain the code.

One important limitation is that only one class can be set in `UserData` field, but it can be easily overcome by defining a new class as an aggregate of more classes. Another limitation is

reentrancy: to eliminate this problem the use of a temporary variable should be avoided: just instantiate the class in the argument of `set_param`. In such a way, every time `InitFcn` is called, a new instance of the implementation class is created. When more than one custom block of the same type is added to the same model there is no ambiguity on the implementation class, so no problem of reentrancy exists.

4. Conclusions

Custom blocks in Simulink can be developed in M language in a variety of methods, but no direct support exists to develop a custom block as Matlab class. In this paper we have shown how object-oriented approach to the development of custom blocks in Simulink is possible. Classes can be stored in `UserData` field in `Simulink.MSFcnRunTimeBlock` run-time object and be available in each callback, where calls to appropriate class methods can be made. Object-oriented approach to custom block development can ease the transition from Matlab code to Simulink, without re-writing of existing code, and provide all the benefits from object-oriented programming to Simulink.

5. References

[Gamma1995] Gamma, E., Helm, R., Johnson, R, and Vlissides, J.. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.

[Lee2004] Lee, E. and Neuendorffer, S., Classes and Subclasses in Actor-Oriented Design. Invited paper in Proc. of the Conference on Formal Methods and Models for Codesign (MEMOCODE). San Diego, California. June 22-25, 2004

MATLAB Documentation

Simulink Documentation