01 Oct 1992

Object-Oriented Modeling of Communication Systems

Kurt Louis Kosbar
Missouri University of Science and Technology, kosbar@mst.edu

Kevin W. Schneider

Recommended Citation

K. L. Kosbar and K. W. Schneider, "Object-Oriented Modeling of Communication Systems," *MILCOM '92 Conference Record - Communications - Fusing Command, Control and Intelligence*, pp. 68-72, Institute of Electrical and Electronics Engineers (IEEE), Oct 1992.
The definitive version is available at https://doi.org/10.1109/MILCOM.1992.244090

OBJECT-ORIENTED MODELING OF COMMUNICATION SYSTEMS

K. Kosbar
Department of Electrical Engineering
University of Missouri - Rolla
Rolla, MO 65401

K. Schneider
Adtran Inc.
4955 Corporate Drive
Huntsville, AL 35805

ABSTRACT

Conventional communication system simulation programs and packages are written using procedural programming languages. Newly developed, object-oriented languages offer the simulation designer significantly different options and structures. By exploiting these new techniques it is possible to significantly increase the flexibility and extensibility of the simulation package. This allows the system analyst to efficiently re-use complex simulation code and quickly and reliably reconfigure the simulation. In addition, a single object-oriented simulation can be used in all stages of the design process, from conceptual design through fabrication and testing. A final benefit of the object-oriented techniques is that the simulation code closely matches the graphical user interface used in most modern simulation packages. This work discusses the basic attributes of an object-oriented model and examines why this may be an attractive simulation architecture.

I. INTRODUCTION

Simulation of communication systems is a reasonably mature field. Several technical journals have been dedicated to this subject, high level computer aided design packages are commercially available, and textbooks covering the topic are now in print. While these are all valuable research tools, many address scientific rather than engineering problems. By this we mean they encourage the system analyst to focus on solving a single, narrowly defined problem, such as determining the symbol error rate of a digital communication system. The analyst typically develops and exercises an algorithm that solves this problem, then summarizes the results in a technical report. The computer program is then archived and the report used to guide the remaining design effort.

However, system analysis is merely the first of many steps required to produce a device. There will be numerous design decisions made, and modifications to the system proposed, well after the high level design is completed. The system level engineering effort will ideally focus not only on answering the immediate design questions, but also on establishing a structure which others can use to further refine and test the system. This structure should place a minimal number of restrictions on the algorithms used to solve the design problems faced at any given stage. Such an approach is even more important in concurrent engineering, where a team of designers address many engineering concerns simultaneously. The unifying structure in these projects could very well be the computer simulation program.

At the most fundamental level, a computer program is defined as a set of data structures and algorithms. Since simple simulation algorithms often place a substantial computational burden on the computer resources, analysts are accustomed to working with complex, but numerically efficient, algorithms. In an attempt to shorten the execution time, simulation research frequently focuses on algorithm development. Constructing a simulation that can utilize these highly efficient algorithms and is flexible enough to address the engineering problems mentioned above will require a more elaborate program. In particular, the simulation designer will need to use both advanced algorithms *and* advanced data structure concepts.

While the exact implementation of the simulation data structure will vary from one application to the next, there are well documented strategies for developing efficient and flexible data structures. One of the strategies that is currently receiving considerable attention is object-oriented (OO) modeling. Individuals that learned to write programs in more conventional languages such as FORTRAN, Pascal and C, must overcome a substantial learning curve before they can effectively use OO techniques. Since most engineers face stringent scheduling and fiscal constraints, they must be convinced that OO modeling holds substantial promise before they will allocate the resources necessary to develop a new simulation using this approach. Fortunately, the flexibility of an academic research environment allowed the authors to investigate the OO simulation problem although at least one of them was initially skeptical that it held substantial promise.

We can now report that there are indeed substantial benefits to OO modeling of communication systems. The intent of this work is to briefly outline these benefits. Section II describes the basic concepts used in OO modeling while Sections III and IV focus on the advantages of the new approach when used to model communication systems. This discussion will hopefully assist others who are deciding if they wish to gamble on developing OO simulations.

0068 0-7803-0585-X/92 $3.00

II FUNDAMENTALS OF OBJECT-ORIENTED MODELING

The earliest high level computer languages, such as FORTRAN focused almost exclusively on algorithm development (recall that the name of the language is an acronym for formula, a.k.a. algorithm, translation). Although extremely elaborate algorithms can be implemented in FORTRAN, only the most fundamental data structures, such as simple variables and arrays, can be easily realized. The next generation of computer languages, including C, Pascal, and refinements of FORTRAN, partially rectified this problem by allowing the user to easily define abstract data structures. While this is a significant step forward, these languages do not incorporate advanced techniques for manipulating user-defined data structures. This can make complex code very difficult to develop, maintain and modify. Recently developed OO languages such as C++, Smalltalk, and Eiffel attempt to remedy this situation by allowing more advanced data structure definition and manipulation.

While the syntax of the OO languages mentioned above differ, the underlying concepts are very similar. Expressing a simulation architecture using these common concepts is the intent of OO modeling. This activity is important enough that attempts are underway to standardize the method of referencing objects (see the Information Object Class specification in the 1992 ISO standard Abstract Syntax Notation - ASN.1). An object oriented model can be implemented in virtually any general purpose language, including machine specific assembly code, conventional languages or OO languages. Since the syntax and structure of an OO language is highly correlated with the OO modeling constructs, it reasonable to use these languages to implement OO simulations.

Three concepts that are fundamental in OO modeling and programming are encapsulation, inheritance and polymorphism. Each of these is briefly discussed below.

ENCAPSULATION

Object oriented languages extend the concept of a data structure to form a new program element which is often called an object. An object contains both data structures and the algorithms that manipulate this data. These algorithms are often in the form of functions and subroutines that are collectively referred to as the object's methods. (To keep the new OO vocabulary at a minimum, we will refer to the methods as functions.) The act of grouping the data structure and the functions into a single program element is called encapsulation. This single program element can then be allocated and used just like any other data type, such as integer or real. Variables can be defined to be of this type. This ability to create multiple copies (or instances) of an object is extremely useful. Each instance has the same data structure, the same functions, but may have different values assigned to the elements in the data structure. This ability to have many copies of the object present in the program is very useful in system modeling where multiple copies of components (such as filters) are often present.

The functions of an object have easy access to the attributes of the object. In C++ or OO dialects of Pascal, the functions have access to the attributes as if they were passed as variable parameters. That is, they can be modified by the function and retain their values after the function is completed. This easy access to the attributes does not require the attributes to be listed in the function statement, so the possibility for mismatch of types is reduced. In addition, since the functions and the attributes belong to the same object, we are assured that the function is working with the data that it is supposed to work with and some other data has not been passed to it by mistake.

One might argue that encapsulation can be carried out in a non-OO language. This is only partially correct. While it is true that functions can be associated with data in other languages (in FORTRAN for example, by using named common blocks), the many advantages of encapsulation as a part of OO modeling can not be exploited. First of all, in a non-OO language, encapsulation is at best an arrangement of the source code. In an OO language, encapsulation produces a unique program element. But perhaps most importantly, the ability to create multiple instances of an object differentiates encapsulation as a part of OO programming from non-OO encapsulation. The ability to create multiple instances without having to copy the source code and rename variables, make OO modeling very useful when modeling communication systems.

INHERITANCE

Often, a group of objects will have several attributes and operations in common. In this case it is desirable to define a core object type that incorporates these common elements and let the various objects in the group include this core with a single program statement. A property of OO languages known as inheritance allows this. Inheritance allows a new object type to be defined that includes all of the attributes and functions of the previously defined object type. The new object type will typically have some new attributes in addition to those that were inherited. The functions of the new object type will include those that were inherited and any additional new functions that are necessary to carry out the operations of the new object type.

In simulation modeling, inheritance can be used to define a base class for all system models that includes all of the

attributes and operations necessary to be a part of the simulation architecture. This object type could be referred to as the "simulation interface". Since these basic functions can be inherited, a programmer working on a system model does not have to be aware of all of the details of the implementation of the basic functions. In addition, if the operation of these basic functions needs to be changed, it can be done in one place and then the new version of the object will be inherited by the other objects. In contrast, without inheritance, the "simulation interface" would have to be modified in each module in the system library. Since a system library could consist of many modules, this would be a very large undertaking.

In addition to its use of defining a base object type, inheritance can also allow a program to be extended without modifying the existing code. Objects from a working simulation can be inherited to form new objects that include some addition or modification to the original object. This allows changes to the system and simulation to be tested without jeopardizing the existing program.

POLYMORPHISM

Inheritance is made even more useful by the use of polymorphism. Polymorphism allows each new object type to redefine the operation of inherited functions as is appropriate for its own purposes. This allows a basic interface to be defined at level of the base simulation object type and then as system model objects are built by inheriting the base object type, the implementation of the functions that make up the interface can be redefined to be an operation specifically for a particular object type.

III. FLEXIBLE SIMULATIONS - MANAGING COMPLEXITY

As the design of communication systems has become more complex and incorporate custom VLSI circuits, the development of hardware prototypes to help make decisions about the design of the system hardware has become an difficult task. As system designers seek to reduce the time required to develop system components, computer simulation becomes attractive as a design tool. When using simulation as a design tool, it is important that the simulation architecture can easily be altered, to handle the many changes in the system design that occur throughout the design process. In these cases, the time that is required to develop the simulation models of the system components is significant when compared to the time required to run the simulation. The many features of OO modeling and programming can significantly reduce the time spent in model development. In this section we will briefly highlight some of the advantages that OO modeling brings to the system simulation model development process.

In a previous section, the property of encapsulation was introduced. Encapsulation allows the simulation model of an object to more closely resemble the "real world" device that it is modeling. Most "real world" devices can be modeled by noting their attributes (or states, memory, etc.) and their operations. These attributes form the data structure of the object that models the device and the operations form the functions of the object. The ability to create multiple instances of objects allows the models of the system components to be used to represent many components of the same type--such as filters or oscillators, that often are present in multiple quantities in a single device. The structure of the object lends itself well to the use of a graphical user interface (GUI) to the simulation. Although OO modeling is not required to use a GUI, the use of OO modeling allows the program to have a structure that is similar to the GUI, thus improving the reliability of the simulation.

As previously mentioned, the use of inheritance and polymorphism allows the basic simulation interface to be mostly hidden from the system model programmer. This basic interface could include "housekeeping" duties such as saving or loading an object's attributes, an initialization procedure and a method to reset the device to an initial state. By using polymorphism, each of these functions can be defined at the base simulation level, but most will have to be redefined with each specific object type, as the objects gain additional attributes and functions. Since all simulation model objects are derived from this base object type, the simulation objects can be passed to functions and subroutines as if they were data structures. The functions of the objects can be called by the subroutines that the object was passed to. Polymorphism allows the proper operation to be carried out when the function is called.

Inheritance can also be used to build hierarchies of objects that can be used to avoid unnecessary duplication of common coding blocks. Each level of the hierarchy includes features that are common to a certain class of devices. Any time duplicate code can be eliminated, it should, to make the code more readable and reduce the probability of coding errors.

One of the larger problems in using simulation as an engineering tool is making the simulation flexible enough to accommodate different design options. By using OO modeling, libraries of system modules can easily be built up and extended as the design dictates. By letting the program design mimic the physical design, the probability for errors in the simulation model should decrease. Maintaining an

evolving simulation is no small task, but it is made easier with OO modeling.

IV. MULTI-PURPOSE SIMULATIONS

Engineers throughout the design chain use computer aided design (CAD) or simulation tools. Each of these individuals face a different set of problems and is tempted to use the simulation package or methodology that allows them to solve their immediate problem with the least amount of effort. For example, simulation is often employed at the highest, or system, design level. Since the designers at this level must work with a very large number of variables and components, they are forced to make numerous modeling approximations to keep the analysis tractable. Such modeling approximations include the use of idealized filter specifications and mathematically convenient noise source statistics. They may also choose to ignore imperfections such as gain and phase imbalance in quadrature circuits, quantization and recursion noise in digital filters, and impedance mismatches to name a few. Subsystems that do not directly effect the overall performance, such as devices used for testing, calibration and verification, may also be completely ignored. At the end of the system design stage, these individuals will have a clear understanding of the system architecture and performance based on their modeling approximations.

The subsystem design engineer will develop a more highly detailed architectural design for each subsystem. They will consider factors such as ease of fabrication, physical characteristics including size, power, weight and thermal characteristics, and develop procedures and devices to test and verify the performance of the subsystem. The result of this effort will be a device that approximates the function the system level designers desired.

For the design to be successful, these two (or more) groups of engineers will need to effectively communicate their ideas and design decisions. When working on a complex system this is no small feat. The success of this communication is verified during testing. Many misunderstandings and mistakes can be made along the design path. These errors can go undetected until the system is fabricated. When a complex piece of hardware fails to meet the design specifications, it can be difficult to determine the cause of the discrepancy. One will need to integrate the products of the various CAD tools with the observations made of the final system.

There are at least two approaches to solving this problem. The first is to allow each design area to use the CAD tools which they find most convenient and powerful for their particular design problem. No attempt is made to integrate these tools. Design decisions and test results are passed between areas in the form of specifications and technical reports. While this approach will certainly work, it invites many problems and a considerable amount of tedious work. This may also limit the complexity of the system and/or the time required to bring the system to market. It can also lead to frustrations because some CAD tools may not easily accept the format of the test conditions, or produce outputs that can not be quickly and easily correlated with the observed data.

Another method of overcoming the design communication problem is to integrate all of the CAD functions into a common package. While this sounds attractive, some are skeptical that one package will be able to perform every CAD task well. There are many different problems encountered when designing a system. It is difficult to believe the software developer will be able to identify all of these tasks and accommodate them in a single design package. This may be remedied by allowing the user to modify aspects of the package. However the resulting packages run the risk of being very large, complex, and difficult to modify.

A third method of integrating CAD tools would be to allow each user to generate the code they need to solve their immediate problem using a general purpose language. However, require them to operate within a framework that allows others to easily use and modify their code. This is the basic concept behind object-oriented modeling. An example of this approach, is given below.

A subsystem of a communication system contains an energy detector. The high level design of this device is shown in Fig. 1. The conventional approach to simulating this device would be to develop three separate subroutines, one for the first low-pass filter, one for the squaring device, and one for the second lowpass filter. Every time increment the simulation would execute these three routines in order.

Fig. 1. Sample High-Level Block Diagram

Suppose this device will be implemented using digital signal processing algorithms on an application specific integrated circuit (ASIC). If synchronous logic is used, the design will consist of blocks of combinatorial logic and registers or Flip/Flops (F/F) to latch the output of the logic blocks, as shown in Fig. 2. All of the F/F will be clocked simultaneously in this design. Assume Logic Block A corresponds to the first low-pass filter, Logic Block B the squaring device and Block C the second low-pass filter. The original order of execution is now starting to become

inconsistent with the detailed design. In fact we may wish to execute Block C first, then B and finish with Block A, the exact opposite order from the original simulation.

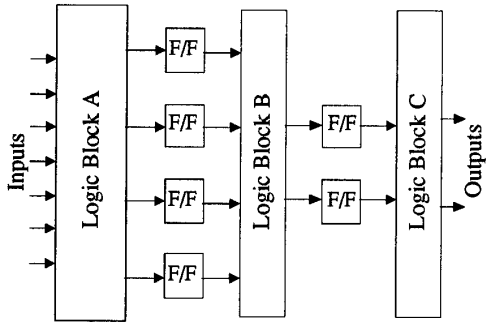

Fig. 2 Boundary Scan
Operational Configuration

The difference between the original, high level simulation and the lower level design becomes even more pronounced when we consider testing techniques. One particularly interesting technique is known as boundary scan [3]. This technique allows one to examine, and if desired, modify, the contents of all of the F/F that fall at a boundary between the logic blocks. This is accomplished by including multiplexing circuitry to allow the ASIC to be reconfigured as shown in Fig. 3. Since all of the F/F now form a single shift register, their contents can be examined or changed by clocking the register the appropriate number of clock cycles. We have now completely destroyed the original execution order. The original subroutine that implemented the first low-pass filter will have to be called twice. This will require substantial modifications to the simulation code.

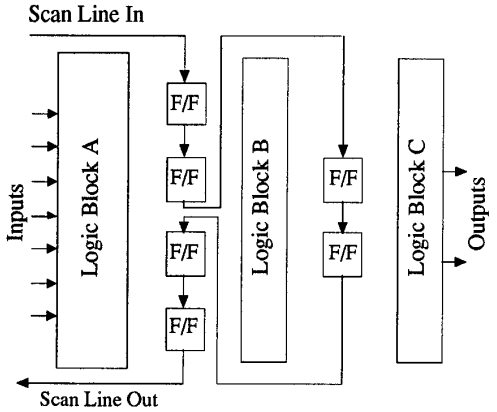

Fig. 3 Boundary Scan
Scan Configuration

In this example, a decision was made to execute blocks in a particular order in the high level simulation. This placed restrictions on the code that ultimately limited its flexibility and usefulness. If the original code had been written so that execution order was not important, it would have been possible to implement the boundary scan technique without rewriting major portions of the code. In this case it would make sense to form an object that was the lowpass filter. Within that object there would be a routine to implement the idealized filter used in the high level design, the actual filter used on the ASIC, and routines to read in and out the values of the F/F to accommodate boundary scan. All of these routines require access to the same data structure. And the data structure should be "hidden" from the rest of the simulation so that it can not be inadvertently modified. By thinking in terms of objects rather than simply algorithms, we could construct a simulation that would be useful to both system designers and logic level design engineers.

IV. AN OO SIMULATION EFFORT

We are currently developing a object-oriented communication system simulation that demonstrates the principles mentioned above. By making extensive use of inheritance, polymorphism and encapsulation the user is able to develop general purpose algorithms within a structure that allows for future change and integration of the boundary scan technique mentioned above. Written in the C++ language, this simulation uses a hierarchy of classes. At the highest level is a "system interface" class that contains all of the housekeeping routines needed to initialize, run and store the results of a simulation, along with graphical interface data. A "wiring harness" class inherits all of the properties of the system interface class, but also allows to user to define a number of state variables. At the lowest level of the hierarchy is a "module" class which inherits all of the properties of the harness class, but also allows for functions that modify state variables. The simulation has many of the attributes listed above. For example, it does not require the user to specify the order in which modules will be executed and can easily incorporate boundary scan.

REFERENCES

[1] G. Booch, *Object oriented design with applications*, Redwood City, CA, Benjamin/Cummings, 1991

[2] J. Rumbaugh et al., *Object-oriented modeling and design*, Englewood Cliffs, N.J., Prentice Hall, 1991

[3] M. Levitt, "ASIC Testing Upgraded", *IEEE Spectrum*, Vol. 29, No. 5, May, 1992, pp. 26-29