DOE/ER/40537--77
CONF-950534--7

An Object Oriented Software Bus

F. McGirt
Physics Division, Los Alamos National Laboratory, MS-M715, Los Alamos, NM 87545

and

J.F. Wilkerson
Department of Physics, University of Washington, Box 351560, Seattle, WA 98195-1560

Abstract

The current techniques of Object Oriented Software Development (OOSD) provide a methodology to develop a set of data acquisition and analysis software tools according to a common specification, in a manner that is analogous to the way computer hardware has been developed since the advent of bus structures for minicomputers and microcomputers.

This approach, here called the Object Oriented Software Bus (OSB), allows one to write independent Object Oriented Programming (OOP) based software objects that correspond directly to hardware objects, data acquisition tasks, and data analysis tools. Software objects have been written for numerous CAMAC, GPIB, and VME based hardware modules. These software objects can then be utilized in acquisition task objects to meet a specific experiment's requirements. This OSB model has been used successfully in numerous small laboratory-scale data acquisition systems and in several large projects as well. A sample of those projects include a neutron beta-decay coincident experiment, an adaptive processing and fuzzy logic support system, and a remote counting system for the Sudbury Neutrino Observatory neutral-current detectors. We will review the general ideas of the OSB method and present some specific examples.

I. INTRODUCTION AND BACKGROUND

For years many individuals in a variety of fields have searched for the ultimate tool that would aid in the development, implementation, and maintenance of large programming tasks. The current techniques of Object Oriented Software Development may provide such a tool by supporting a way to develop software according to a common specification, similar to the way hardware has been developed since the advent of bus structures for minicomputers and microcomputers. This new approach is called the Object Oriented Software Bus (OSB).

A. Scientific Computing

The decade of the 1960s was a blossoming period in scientific computing in that for the first time large mainframe-type computers were available and were being applied to the solution of problems that were previously felt to be too complicated or too time-consuming for pure human solution. During this time many specialized numerical methods and techniques were developed that were especially suited for use in large-scale computerized numerical experiments. Examples of these were the Monte Carlo method which predicted particle transport through various media and the Sn method which was widely used for calculating neutron flux in nuclear reactors. Indeed each field of science and engineering developed their own rather extensive set of computerized techniques especially suited to their own interests.

Nearly all of these scientific programs were written in procedure based languages such as FORTRAN. The focus tended to be on processing and algorithms. Actions to data were accomplished by passing the data from one subroutine to another. Needless to say these efforts resulted in the development of some very large computer programs which soon were found to be very unwieldy and almost impossible to be understood by one person. Hordes of programmers and scientists were then organized into project teams in efforts to keep such programs running and also to add new required features.

In the period of the 1970s and the early 1980s controversy raged among the computer scientists and programmers about how best to manage both the large software development tasks (the actual coding techniques) and the large number of programmers that were then felt to be needed to accomplish the required work. The problem, of course, was how to break up a large programming task into smaller pieces each of a size that could be handled by one or a few programmers, and then when each piece was complete to have it fit with the other pieces to successfully accomplish the intended task. What evolved was "jigsaw puzzle" programming at its best. Not only would the programmers responsible for a particular piece of the puzzle have to ensure that their piece performed its intended task correctly, but they would also have to make sure it

DISCLAIMER

interfaced with all its adjoining pieces exactly. The communication difficulties among the human project members alone made the construction of successful programming interfaces difficult. Numerous techniques such as "structured programming", "modular programming", "structured analysis", etc. were invented as were new programming languages such as Pascal, C, Modula, etc., in an attempt to improve the life of the hapless programmer. Some of these new methods helped a great deal, but none offered a solution to all of the problems.

B. Data Acquisition and Control Computing

Analogous to the evolution in scientific computing, in the 1960s scientists and engineers began to realize that computers also would be very useful both in controlling critical parts of laboratory experiments and in recording the data from those experiments. No longer would the experimenter have to sit with the experiment for long hours tweaking control parameters and filling up page after page of laboratory notebooks - the computer would perform such tasks in a much more reliable fashion. However the mainframe computers of the day such as the IBM 7094 with their bulky peripheral equipment were in most cases much larger and more expensive than the actual experiment to be controlled.

Fortunately, Digital Equipment Corporation (DEC) in the mid 1960s began to market their PDP line of small computers which soon became known as "minicomputers". These devices such as the PDP-6, the PDP-8, and later the PDP-11 began to appear in major accelerator and nuclear reactor laboratories across the US and Europe. As these minicomputers were quite affordable by small academic institutions, computer control of even undergraduate-level laboratory physics experiments became commonplace.

Along with the development of the minicomputer came a second new industry. Many new companies were formed whose sole purposes were to supply accessory hardware for the minicomputers such as new input-output devices. But how was this possible? The hardware architecture of each of the mainframe computers of the day were all different so that it was almost impossible for a company other than the manufacturer of the mainframe computer to even supply a printer for another mainframe. The answer was the hardware bus which the minicomputer used to interface to its peripheral equipment.

II. HARDWARE BUSES

A. The Computer Bus

The concept of buses was not new when minicomputers came along since buses had been used for years to link components in the electrical power industry; what was new was the concept by DEC in the design of the minicomputer to more clearly separate the function of the computer's central processing unit (the component that executed the logical and arithmetic functions of the computer) from the peripheral functions of input-output such as keyboard terminals, paper tape devices, and printers. A second but equally important new concept by DEC was to extensively (for that time) document and publish the specifications of the hardware peripheral interface. These specifications included not only the electrical signal levels and states, the timing requirements, etc., but also instructions and hints on how one could easily build hardware devices to perform other specialized functions not supplied by DEC. Soon this hardware interface to the minicomputer and its associated specification documentation became known as a "computer bus".

Unlike the situation with the mainframe computer community with its "out-of-control" hardware and software development environment, the minicomputer developers were able to more easily break up large tasks into small ones and to give these smaller tasks to many people with a better assurance of the separate pieces fitting together in the end since there was a common specification for development - the computer bus. Now for the first time one could purchase a minicomputer from one company, a printer from a second company, and even memory for the mini from a third company. Indeed these individual manufacturers of minicomputer components thrived and greatly improved the capabilities of the end user computer system.

B. The Interface Bus

As the sales of minicomputer peripherals increased, the manufacturers of these peripherals began to recognize the potential market for many other interface components. Among the market possibilities were the scientific and military R&D communities and their desire to build and use more and more complicated experiments and acquisition systems. What was lacking from the manufacturers viewpoint was any clear specification of a common interface to all of the many different experiments or systems - it would be very nice to be able to build a single digital input-output interface or a single analog-to-digital converter interface that would be useful for hundreds of different applications rather than to have to build hundreds of different interfaces each tailored to a single application. A solution to this problem appeared in the form of the interface bus or a definition similar to the computer bus which allowed the computer to also interface to the "real world" rather than just to its own peripherals. One of the first of these was the specification of the Computer Assisted Manufacturing And Control (CAMAC) interface bus which has since seen widespread use in physics and military applications. The Hewlett-Packard company invented an interface bus known as the Hewlett-Packard Interface Bus (HPIB) which evolved into the industry-standard interface bus used for computer interfacing to test instruments such as voltmeters and frequency counters. It is now known as the General Purpose Interface Bus (GPIB). The trend today is to

form consortiums with industry, academic, and other interested representatives and then to develop new interface bus structures that attempt to meet the needs of all the consortium members. The current VME and VXI interface buses were both developed by such processes.

III. Object Oriented Software Development

As discussed above the evolution of the computer mainframe industry and the minicomputer/microcomputer industry has proceeded along different lines with respect to hardware development. The minicomputer/microcomputer industry has made available from the beginning cross-manufacturer standards or bus definitions which have enabled many companies to produce competitive products of similar function to the benefit of all concerned. Unfortunately, the scientific and engineering communities' software development process has not kept pace with the evolution of hardware standards and in fact the software process is still argued by many to be "an art" which can never have sound engineering principles applied to its development. However, the recent widespread availability of languages that support Object Oriented Programming methods and the concurrent maturation of Object Oriented Software Development (OOSD) has made that argument much less valid.

IV. Object Oriented Programming

Similar to other software methods, the OOSD process can be grouped into three related areas: analysis, design, and implementation. However, OOSD is distinguished by the reliance on the concept of "objects" and by the use of Object Oriented Programming languages to implement these objects in code. One of the chief advantages of an object approach is that it allows one to manage the complexity of large intricate programs with software objects that correspond to natural real world objects. However, there are also other benefits.

A. Encapsulation

The basic difference between conventional programming techniques and Object Oriented Programming techniques is the way in which the data and the actions on that data are treated. In conventional programming the data and all actions on that data are treated as two separate entities. One defines the required data structures, arrays, data common blocks, etc., and even perhaps the data flow throughout the program; then one separately defines the actions (or functions) to be performed on the data. Obviously for each new data structure defined, new functions must be defined to performed the required actions on that data.

In Object Oriented Programming, both data and actions on data are handled as one encapsulated entity. An object is in fact defined to contain data **and** an associated set of actions that operate on that data. Therefore when the data are defined, the actions on that data are also defined. Instead of a set of functions that act on the data there are a set of objects interacting with each other.

B. Messages

Objects interact with each other by sending each other messages. In order for say object A to make object B perform an action on object A's data, then object B is sent a message by the object A calling for the action to be performed. For example, an object might be created that represents a rectangle. Its data structure contains the location of the four points of the rectangle. Its actions might include drawing the rectangle, erasing the rectangle, and moving the rectangle. To draw the rectangle on a terminal screen, a draw message would be sent to the rectangle object. The details of drawing the rectangle would be fully encapsulated within the rectangle object.

C. Inheritance

Other than encapsulation, the other and more important benefit of Object Oriented Programming is a property that objects have of being able to inherit data and actions from other objects. Every object is a member of a class of objects. A class definition describes the object's data structures and the messages to which it may respond. In addition, subclasses of objects may be defined which may be slightly different from their superclass but inherit all of the data structures and actions of the superclass. For example, an object that represents a square may be defined as a subclass of the above rectangle class. The square object would inherit the data structures and inherit and respond to the same messages as the draw, erase, and move actions of its rectangle object superclass, but would also have its own data structures which could specify that it is a square rather than a rectangle. It could also respond to additional messages such as a block fill of the interior of the square. This property of inheritance of objects is the primary reason for the capability to reuse previously written (and debugged) objects in more than one application.

V. Examples Of Objects For Practical Application

The use of OOPs techniques has improved the state of software development for practical data acquisition and control applications in three major areas: data analysis, user interfaces, and hardware interfaces.

A. Objects For Data Analysis

In supporting data analysis work one is able to create different kinds of objects that represent different tasks to be performed; each object is fully encapsulated with its data structure that it must operate on; and these objects may be

reused over and over again in other data analysis programs. If the data requirements for those programs are different, subclasses of the existing objects may be written with only the new differences being specified. If new or different algorithms are required, existing objects may also be subclassed with only the new differences being specified. Examples of data analysis objects include a point plot object, a histogram object, a FFT object, a Monte Carlo object, etc. Companies are now also beginning to develop and market data analysis libraries of objects which may shortly replace some of the now popular conventional numerical function libraries that require understanding of the many function parameters in order to be used successfully.

B. Objects For User Interfaces

The subject of user interfaces has always been a source of heavy debate in software development circles as one strives for the ultimate "user-friendly" operation. For data acquisition and control applications the user interface can also have an additional effect because of the sometimes critical nature of the hardware or tasks being managed by the software. One often spends major amounts of time and code resources in the attempt to ensure that the user interface is as robust as possible and that all interactions between the user and the system are completely validated. The recent increased use of Graphical User Interfaces (GUI) has improved the situation for the user, but not for the developer, since even slight modifications of the GUI dialogs can have a potentially adverse effect on the expected operation of the program.

This situation can now be vastly improved with the use of OOPs techniques. As in the case of data analysis objects, companies are marketing complete user interface object libraries such as Symantec's Think Class Library and Metrowerks' PowerPlant. These libraries contain objects for virtually all items required for user interface dialogs - buttons, text edit fields, file requests, popup menus, etc. Since these items are implemented as fully encapsulated objects they may be manipulated in exactly the same way as the data analysis and hardware objects discussed above. Therefore, one may now change user interfaces with little effort knowing that these changes will have no adverse effects on any other part of the program.

C Objects For Data Acquisition And Control

For data acquisition and control hardware interfaces, the situation is even more improved. Hardware modules such as bus controllers, bus interfaces, bus-resident processors, test instruments, data monitors, etc., along with their associated configuration data structures and required control tasks may now be represented as true OOPs objects. An experimenter may now choose from a set of CAMAC, VME, VXI, etc., hardware modules with each one of these modules accompanied by its own fully encapsulated support software -including data structures. In addition to being able to use off-the-shelf hardware components in a standard hardware bus environment, one is now able to also include off-the-shelf software support in object format for each hardware component. This software is developed and debugged once and may be reused in many applications.

VI. AN OBJECT-ORIENTED SOFTWARE BUS

The result of the successful use of Object Oriented Programming techniques in both data analysis and data acquisition applications encourages one to envision the future development of an industry standard object oriented software bus environment very similar to the standard hardware bus environments that evolved along with the minicomputer and microcomputer development processes. If this occurs, then software development will perhaps truly become an engineering activity as it properly should be. Indeed the hardest task for most software developers may well be to unlearn much of what is known about traditional programming practices.

A specific example of a standard object oriented software bus has been developed to provide a starting point for future data analysis, data acquisition and control applications and to display demonstration techniques for those who wish to assemble their own applications from current off-the shelf objects or to include objects of their own design. Using this example as a model, laboratory-scale data acquisition tasks may be created and executed in days, rather than weeks or months. Although this general example contains a substantial amount of code, all of it is encapsulated in reusable (and hopefully fully debugged) objects.

A. OSB Implementation

The following is a brief summary of the implementation of an OSB-based data acquisition system.

A.1. "Modules"

In this OSB system each individual hardware device has a corresponding OOP software object. All of these objects are grouped under a category of "Modules". Each module/object has 3-4 associated functions: configuration, basic operations, special operations, and in some instances utilities. Modules fully support only one piece of hardware. There are a number of classes of modules, including controller and I/O types. Since modules support all possible hardware operations, they are ideal for testing and debugging the individual hardware devices.

A.2. "Tasks"

"Tasks" are objects that utilize one or more hardware devices in a coordinated fashion to acquire data. Tasks usually perform initialization of the hardware, control of the acquisition process, and finally recording and/or display of

the data. In some instances the direct control of the hardware is taken care of by an embedded processor(s) located in the VME. In these cases, the Task downloads the software to the embedded CPU, starts the processor running, and simply transfers data between the VME hardware and the user computer.

A.3. "Analysis"

Analysis objects allow one to display and manipulate the data. Examples of such objects were discussed in Section V, subsection A.

A.4. General Code Architecture

The OSB architecture allows - within a module object - the selection of which hardware controller hardware (object) is going to be used at run time, with no recompile of the code. In addition, the OSB code was designed and written to make the procedure of adding duplicate or new modules and tasks very simple. Adding duplicate modules is supported at runtime. When adding a completely new task or module the programmer simply has to modify a few lines in a configuration file. All GUI related details for that module or task are then included.

B. Supported Hardware

Our OSB system has been developed on Macintosh computers using the Metrowerks CodeWarrior C++ compiler (earlier versions used the Symantec Think C OOP compiler). Hardware buses that are supported include CAMAC, GPIB, VME, and NuBus. The Macintosh communicates with these links via dedicated NuBus based parallel controllers (bus adapters) or ethernet connections. Supported configurations include single crate, multi-crate, and mixed CAMAC-VME-GPIB systems. We also support the MVME 147 and MVME167 embedded CPUs (currently in standard C code). A variety of hardware modules have been implemented as objects.

C. Current Examples

The object oriented software bus model described above has been used successfully in numerous small laboratory-scale data acquisition systems and in several larger projects. A sample of these projects include a neutron beta-decay coincidence experiment, an adaptive processing and fuzzy logic support system, and a remote counting system for the Sudbury Neutrino Observatory neutral-current detectors. In each of these applications the software bus approach allows one to quickly understand and verify the operation of the individual hardware components (as objects) and then to easily assemble those components to form the final system. The development of objects for new hardware modules that each system required was also done in a very short period because the object oriented software bus allowed one to concentrate on the actual hardware functions of these new components. Using the old development methods one would have spent a large amount of time in validating the new user interfaces, determining the possible effects of adding new software to an existing system as well as dealing with the new hardware features and functions of the new modules. Finally, since two of the projects required some of the same new objects one only had to develop those new objects once and could then use them without any modification whatsoever in the second project.

D. Future Work

Most of our efforts have concentrated on the development of hardware modules and tasks for data acquisition, with only simple analysis tools being supported. We are now in the process of extending our data acquisition frame work to allow the simple inclusion of analysis tools (in the exact same fashion as we allow the addition of new Modules and Tasks). We are also adding more sophisticated display features, including a set of C++ based histogram tools.

We anticipate that future advances by commercial vendors of multi-platform compiler support will allow our code to be used on a variety of end user computer platforms and buses.

VII. SUMMARY

This paper has described a new approach to the development of software for highly integrated software - hardware systems such as the type used for data acquisition and control. This new approach is called the Object Oriented Software Bus and is a way to develop software according to a common specification similar to the way interface hardware has been developed since the advent of bus structures for minicomputers and microcomputers. The key concept of the OSB is the extension of the common use of objects to support user interface and data analysis functions to the development of software objects that directly correspond to real-world hardware interfaces and modules.

The authors welcome discussion on this topic and encourage its development toward a possible future industry standard.