Proceedings of the Seventh Annual National Conference on Ada Technology

March 13-16, 1989

Sponsored by
ANCOST, INC.

With Participation by
United States Army
United States Navy
United States Air Force
United States Marine Corps
FAA Technical Center

Co-Hosted by
Monmouth College
Penn State/Harrisburg
Jersey City State College
Cheyney University
Stockton State College

REUSABLE SUBSYSTEMS FROM A HIGH PERFORMANCE ADA COMMUNICATION SYSTEM

Thomas L. Chen & Walter Sobkiw
ECI Division, E-Systems, Inc.
St. Petersburg, Florida

ABSTRACT

The reuse of functionally equivalent software is limited by performance and reliability requirements. The reusability can be improved when the software system is designed for each class of applications following requirements established for a reusable architecture. The reusable system is made up of functional objects and binding objects that follow a set of program paradigms. The functional objects and binding objects in a class of applications are mixed and recombined to achieve the best performance and reliability according to the hardware and operating system used to drive the application.

INTRODUCTION

The data communication industry, more than any other industry, is obsessed with standards and conventions. It can therefore be expected that there is a high degree of reuse of existing software in this industry. There is indeed a very high degree of reuse in this industry. This is evident by the popularity of SNA and DecNet. However, there is a continuous stream of communication software being developed from scratch. This is especially bothersome because most existing network software facilitates the installation of custom protocols where it is required.

The need for custom communication software is justified by the performance and reliability provided by existing software - on the hardware dictated by the application - which does not meet the requirement of the intended application. The reusability of existing software is then limited by the performance and reliability when it is installed on a given set of hardware. This paper presents an approach that manages reusability and portability for high performance data communication software.

CURRENT SOLUTIONS

There are two current solutions where software reuse has been successful. The first solution uses an existing data communication software system, augmented by custom protocols, that satisfies the performance and reliability require-ments on the hardware and is acceptable to the functional applications.

The first solution is most desirable as long as the hardware required to deliver the required performance is not limited by:

- Space
- Cost
- Reliability
- Weight
- Power consumption
- Processing speed

The second solution uses existing subroutines or small software packages to support reusability for a new functional application. This solution offers very little saving because the majority of the software cost is in the design, integration, and system test. Most of the cost is not in coding and unit test.

Another possibility is the reuse of software subsystems from existing software systems to build high performance software for specific applications. Approaches for using software subsystems as opposed to whole software systems or small software units to support reusability have not been extensively studied. The advantages, disadvantages, and problems associated with this third approach are not well known.

CURRENT APPROACHES TO INCREASE SOFTWARE REUSE

The current design methodologies, whether they are structured or object oriented design, approach the software reuse issue on the following two principles:

1. Identify common functions through implementation independent functional decomposition or object identification.

2. Encapsulate the implementation of the common function, discussed by Cox and Hunt [1].

There has been a fair amount of success with this approach. However, 15 years after the introduction of structured system analysis, the software reuse problem is still a subject of significant study.

LESSONS FROM SUCCESSFUL REUSABLE SOFTWARE EFFORTS

There are many successful systems where parts of the system are rearranged, or augmented with new software, to form different applications. Most notable are:

1. UNIX PIPE for text file applications

2. Transaction processing systems like CICS

These reusable software systems have the following common characteristics:

- The software is only reusable in a Compatible class of applications. The different applications in the same class can be as diverse as an air cargo system or a bank debit system.

- A defined Binding Architecture that spells out the different subsystems comprising the system and defines how each piece should fits.

- Binding Objects that tie all pieces together but do not directly add to the functional requirements of the application.

- Functional objects that are directly related to the functional requirements of the application.

- All the functional objects and binding objects are constructed to a compatible Program Paradigm for the unique reusable system.

- Portable Language.

It is interesting to note that these successful reusable software methods were developed before structure programming, structure system analysis, and object oriented design were introduced into the software community.

Figure 1 shows how the hardware community has developed the capability to mix different subsystems into an appropriate system that can support multiple applications. This has been supported with the definition of standard hardware "Binding" objects that permit the linking of various functional objects to other functional objects. An analogy between the hardware and software community is shown in Tables I and II.

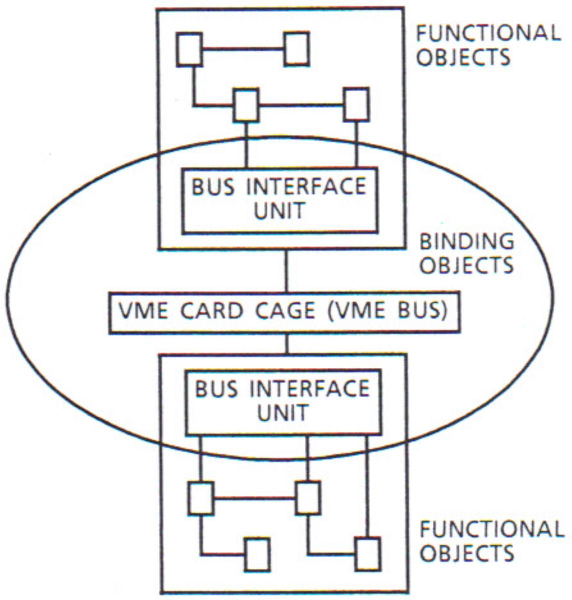

Figure 1. Hardware Reusability

Table I lists the hardware equivalents of the reusable system characteristics. Table II lists the different parts of the reusable software system according to the required characteristics.

Table I. Hardware Equivalent Partition Example

Class of Application	Binding Architecture	Binding Object	Functional Object	Compatible Paradigm	Portable Language
Bus based computer system	Bus specification	Physical bus, pin layouts, and bus interface units	Unique logic on application boards	VME multibus	Bus protocol/ handshake

Table II. Reusable Software System Partition

Class of Application	Binding Architecture	Binding Object	Functional Object	Compatible Paradigm	Portable Language
UNIX text file manipulation	- Text file only - Serial processing	Pipe I/O redirection	Grep, cat, ls, type	Use standard input output device	"C"
On line transaction processing	Transaction processing programming guide	CICS (program, transaction files and configuration files)	Transaction programs	Six step program cycle; Atomic operation transaction driver	Cobol Fortran PL1 *Bal

*Not portable, reusable only.

We can also draw the following observations from these successful software systems about software reuse:

- Easy rearrangement of existing functions to produce new applications is a key for software reuse.
- A narrowly defined class of application is often enough to support the additional cost of software reuse.
- Exact match of function is not necessary for reuse.
- Easy addition of new functions is essential.

REUSABLE HIGH PERFORMANCE COMMUNICATION SOFTWARE DESIGN APPROACH

The novel reusable software design approach described in this paper is based on observations of limited reusability in both the software and hardware community and on two considerations which are not advocated in current software engineering practice.

The first consideration is acknowledging that the total software solution includes extensive amounts of software executable code used to support the binding of the functional application software to each other, to the hardware, and to the operating system services. In this novel design process, there is a conscious effort to separate the purely functional pieces of software from all other software that is dependent on the hardware and operating system environment.

The second consideration is that the binding effort and the selection of the hardware is not a one time event in the life cycle of a software project. This is especially true in the high performance embedded system. This point was expanded upon in a discussion about fault tolerance and performance by Chen and Sobkiw [6]. Thus, if an effective mechanism could be developed to isolate unique "binding objects software" from "functional objects software," then not only will the potential for reusability increase, but also during the course of software development/modification, the effort may be reduced as functions are bound in different ways to support various stages of development. The functional design of the application and the elaboration of the binding effort, as well as the selection of the hardware, can be carried out as two independent activities if the two interfacing activities are properly defined.

Figure 2 shows that there is an area of software activity that eventually translates to unique code. That software effectively allows the application to become integrated with the operating system and hardware services. This is shown conceptually in Figure 3.

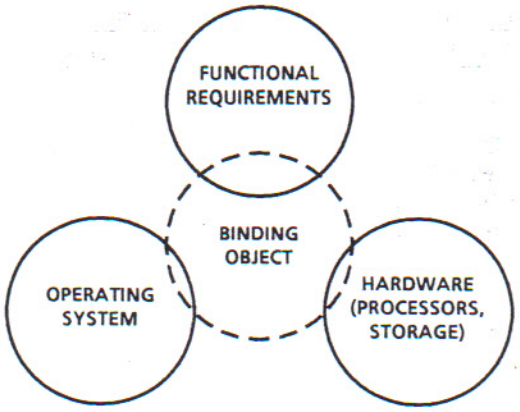

Figure 2. The Software Activity

A major piece of software is overlooked by today's design methodologies. This software binds the functional software, operational software, and hardware resources.

In structured system analysis the functional design is bound to the operating system and hardware after an implementation independent analysis. The same can be said of the OOD techniques in which the unique hardware architecture is bound with the resulting OOD based design. These design approaches were driven by two assumptions.

The first assumption was that the software design starts with an implementation independent analysis which defines functions and data flows. Then, the software functions are allocated to hardware resources. Each group of functions in a hardware resource can be allocated into software processes and these processes can be designated as a collection of procedures by structured system analysis or other techniques. This one time procedure is seldom successful. The allocation of processes in the data flow diagrams are either done according to the target system at the very beginning or are not used at all when the final software processes are allocated to the hardware. This practice is partly confirmed by Post [4]. Chen and Steimle [9] illustrate the drastic differences in the software design that performs the same application function but delivers different performance characteristics. A major portion of the software design is unaccounted for in the solution. The unaccounted for software in the design is the software that effectively links the hardware resources and operating system resources to the applications software.

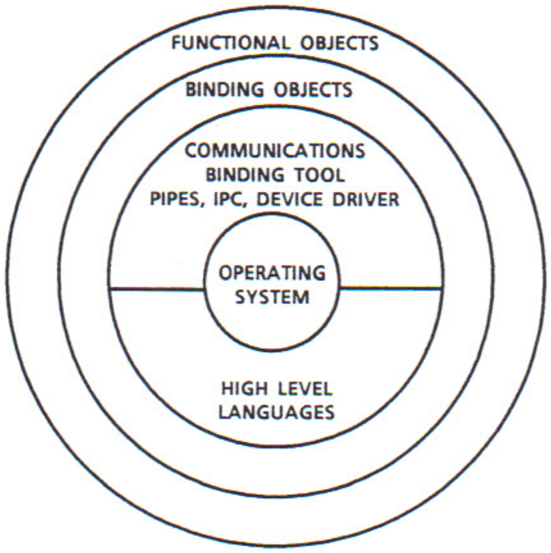

Figure 3. System Layers

The binding objects are not unlike the transaction support systems, provided by UNIVAC or IBM, that address non-transaction oriented activities such as Fault Tolerant communications.

The second assumption was that the software designer does not need to understand the hardware being used in the system. In order to achieve performance, unique hardware and operating system control structures are used in the the final solution. These structures control parallelism, manage storage, address data integrity and other key system characteristics. Karp [8] and Burger [5] elaborate on this point. This discussion on the explicit control of parallel activities and storage management can be defined as a binding effort to mate the application to the chosen hardware.

Figure 3 illustrates how the software in a system can be seen as a layered collection of elements. At the heart of this collection is the operating system which mates all the software to the hardware. Next come the languages, linkers, IPCs, and system configuration files that not only translate application program source code to executable code, but also define the profile of the application and bind the application to the services and facilities provided by the operating system. The application gains the services of the CPU and I/O by manipulating these services. The next layer is the binding objects layer. The outer layer of this collection of elements is the functional objects. The functional objects must follow the interface rules to the inner layer while satisfying the functional requirements of the application.

This picture is not new and there is an existing model for this concept in the form of transaction services. The transaction services of IBM, UNISYS, and other computer vendors allow multiple applications to be developed without recreating the software that links the primary mission applications software to the hardware and operating system services. This shell can be extremely large in terms of the total software effort depending on the system characteristics.

Examples include the transaction processing paradigm provided by Sperry TPS 1100 and CICS supported by IBM. The binding objects are transaction processing support software items provided by the vendor. The transaction programs are discrete programs provided by the user that satisfy the functional requirements of the application. In the case of Sperry, these programs must be coded according to a style defined by TPS 1100 and follow the interface rules to TPS 1100. The same requirements are true for the CICS supported by IBM.

The issue is that if a software IC is to achieve reusability then that software IC should be purely functional in nature and not contain any "glue" to bind it to hardware or operating system services. In other words, the software IC should be separable from the architecture of each application. In addition, the success of a software "IC" is based on its firm, fixed, accepted interface definitions which effectively translate to the architecture of that software IC. The binding objects in Figure 3 must present a standard, well defined, well accepted interface to the functional objects.

The step taken to design this system is a pragmatic one. First, the architecture of the system is laid out to contain all the characteristics of a successful reusable system. Table III lists the architecture components of the reusable high performance communication system according to the required characteristics. Structured analysis, as well as object oriented programming technique, is used to build the functional objects and the binding objects as illustrated by Chen and Sutton [3].

Table III. Reusable High Performance Communication System

Class of Application	Binding Architecture	Binding Object	Functional Object	Compatible Paradigm	Portable Language
High perform-ance software	UNIX processes, distributed hardware dedicated processer for asynchro-nous inputs	IPC, shared memory shared disk file Ada facade	Protocols manage-ment entities	Single select for multiple asynchro-nous inputs interlocked acknow-ledge	Ada

BINDING ARCHITECTURE

Within UNIX a high performance application with asynchronous inputs is made up of UNIX processes and device drivers. These UNIX processes and device drivers can be distributed into various hardware.

The UNIX processes are further divided into input processes and principle processes. Each input process is dedicated to an input. Sufficient input processes are created so that there is always a free input process available when the device driver receives an input from any device. The UNIX processes communicate to the device driver through file I/O. The UNIX processes synchronize with each other through shared memory, IPC, disk files, and interprocessor IPC. The UNIX processes are distributed across several loosely coupled processors.

Each UNIX process must be programmed according to a program paradigm which is compatible to the binding objects and the binding architecture. Each UNIX process in this application is made of the following components:

Binding objects:

- Main program

  This is the software that ties all procedures together to form a UNIX process.

- System Access Packages (SAPs)

  This is a procedure interface to procedures in a different process.

Functional objects:

These are the software packages that directly relate to some application functional requirements.

BINDING OBJECTS

Binding objects are those software items that tie all pieces of the application together but do not directly support application functional requirements.

- Main program
  The main program is the one single part that ties all the packages together when the subsystem is used as a process. This program is individually developed for each process.

- System Access Packages (SAPs)

  The system access point is defined as a software package which is independently developed to connect functional objects in different UNIX processes. Instead of interfacing directly, through IPC or shared memory, the functional objects interface with the system access point(s).

  This concept is similar to the remote procedure call elaborated on by Wilber and Bacarisse [2]. A SAP contains three main parts:

  1. Server interface package - A package specification.

  2. Client interface package - A package specification.

  3. Body objects - Several body objects are required for each SAP. There is one package body for each interface.

FUNCTIONAL OBJECTS

Functional objects are those software items that are directly related to the functional requirements of the application. The functional requirements of a data communication system can be partitioned into the following functions according to the ISO 7 layer model. The Data transportation functions are:

- Application
- Presentation
- Session
- Transport
- Network
- Link
- Physical

The Management functions are:

- Monitor and record
- Network management

Through standardization, each of component of this model is reusable in different applications. There are, however, many protocols. Therefore, each application that wants to incorporate reusable code must blend implementations of the protocols.

A high performance communication software system can be made up of the following two groups of UNIX processes:

Data transport:

- Front-end network process
- Back-end network process

Management entity:

- Monitoring and recording process
- Network management process

The Data transport subsystems have the following characteristics:

- They implement parts or all of the ISO 7 layer functions above level 2 protocol. The two lower level protocols are implemented as UNIX device drivers.

- They transport data from one connection to another connection. The sub-system usually consists of a protocol part and a set of tables describing each connection.

- They can easily be replicated and distributed over many computer processors.

The Management subsystems have the following characteristics:

- They communicate with all 7 layers of the ISO model.

- They either provide a centralized control for the whole communication system, or provide a centralized repository for the whole communication system.

- They can be distributed over many processors only in a hierarchical manner.

In the implementation of a customized communication system, these subsystems are combined, or split into a number of processes, and distributed on the selected hardware (computers) according to the operating system characteristics and the application traffic flow to obtain the best performance for the hardware chosen to host the communication system.

PORTABLE LANGUAGE

Ada is mandated by the contract for this data communication system. Its package features enable an elegant implementation of each SAP, shown in Figure 4. Ada package specification provides a nice Ada facade for each SAP that can be independently compiled. The features provided by Ada are severally hampered because each Ada linked output is implemented as a single UNIX process. The very large load module generated by Ada is also an issue of concern.

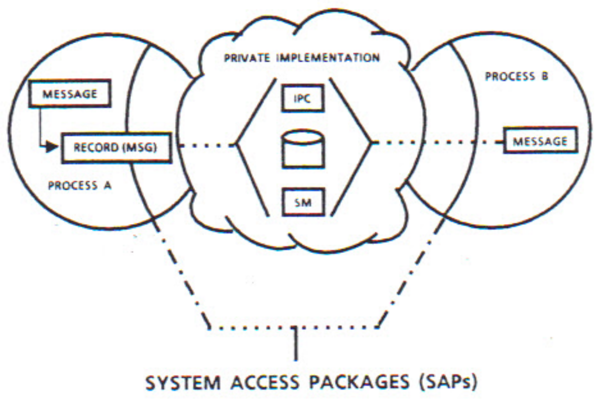

Figure 4. System Access Package (SAP)
The SAP interfaces the functional application to the operating system and hardware environment.

PROGRAM PARADIGM

All the functional objects must be constructed according to compatible program paradigms for this class of reusable system. This programming style is developed in the traditional transaction processing system shown in Figure 5.

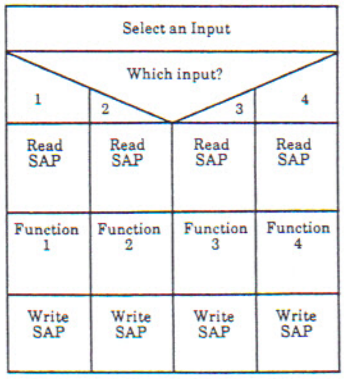

Figure 5. Compatible Program Paradigm Loop

Each transaction program is made up of one or more transactions. The transaction program accepts asynchronous input only in one predetermined location in the program. Each transaction is driven by one unique input. The transaction processes the input, updates related data base information, stores intermediate results or generates output, then loops back and waits for the next input. The programming style can be illustrated by the following example where a free style program is transformed to a transaction program:

```
1.  Read A                 --asynchronous input

2.  Read B                 --asynchronous input

3.  C=:A+B  -----------    Function requires two
                           asynchronous inputs

4.  Write C

5.  Loop
```

A transaction program to accomplish the same requirement looks like the following. The sequence in each column is a transaction.

```
1.  Read A or B           -- Home position wait for
                          inputs

2.  If A                      Else

3.  If B is on queue          3.  If A is on the queue

      then                          then

        get B --                      get A --
        Synchronous                   Synchronous
        input                         input

          C:=A+B                        C:=A+B

          write C                       write C

      Else                          Else

          put A on queue                put B on queue

      exit or loop                  exit or loop
```

Synchronous inputs are inputs that the transaction can get on demand. These inputs are stored in shared memory or in local disk.

CONCLUSION

Our attempt to build a high performance communication system incorporating reusability was marginally successful. We demonstrated that when major functions are rearranged, a specific level of performance can be achieved. We showed that Functional objects are reusable and portable; that Binding objects are reusable but not easily portable to different operating systems; and the use of an enumeration type makes the addition of a new function difficult.

The ECI approach to reusability was attempted on a high performance comm system with approximately 100K lines of Ada code. Although the program did not specifically identify reusability requirements, we did include an effort to identify characteristics which supported reusability.

The design required several physical allocation changes in the development cycle to achieve the best performance and reliability goals. These changes were accomplished with no changes in Functional objects. This provided confidence that the function aspects could be ported to different system hardware platforms to combine with new Binding objects to carry out the same application. This approach to reusability accommodates the different architecture requirements to achieve the best performance and reliability in difficult hardware platforms.

In summary, this novel approach to reusability is based on acknowledging that systems consist of functional and architecture dependent code. Given this assumption, the design process includes separating functional code from architecture code early in the effort and defining a binding mechanism that uses existing services, the SAP and the mainline.

We recognize that our paradigm needs extensive refinement and expansion to provide the level of reusability needed in current Ada applications. ECI expects to continue its research into the applications of reusability in three existing programs, and will attempt to expand its present data base to other systems through several mechanisms now under active research.

REFERENCES

1. Brad Cox and Bill Hunt, "Objects, Icons, And Software-ICS", Byte, August 1986, pg 161.

2. Steve Wilber and Ben Bacarisse, "Building Distributed System With Remote Procedure Call", Software Engineering Journal, September 1987, pg 148.

3. T. L. Chen and M. Sutton, "Object Oriented Design: Is It Enough For Large Ada System", Proceedings of 1988 ACM Computer Science Conference, pg 529-534.

4. J. Post, "Application Of A Structured Methodology To Real Time Industrial Software Development", Software Engineering Journal, November 1986, pg 222-234.

5. A. H. Karp, "Programming For Parallelism", Computer, Vol. 20, No. 5, May 1987, pg 43-55.

6. W. Sobkiw and T. L. Chen, "Design For Fault Tolerance And Performance In A DOD-STD-2167 Ada Project", Proceedings of the Sixth National Conference on Ada Technology, pg 424.

7. R. P. Wiley, "A Parallel Architecture Comes Of Age At Last", Spectrum, Vol. 24, No. 6, June 1987, pg 46-50.

8. T. M. Burger and K. W. Nelson, "An Assessment of The Overhead Associated With Tasking Facilities and Task Paradigms In Ada", SigAda, Vol. VII, No. 1, pg 48.

9. T. L. Chen and C. L. Steimle, "Two Design Approaches Using The Ada Language", IEEE Southeastcon 87, Vol. 1, pg 72.

BIOGRAPHY

Thomas L.C. Chen is a member of the Technical Staff in the Software Systems Department, E-Systems, ECI Division. He is the principle software designer of Survivable Communications Systems, has over 25 years experience in the development of communications methodology. He holds a M.E. from Taipei Institute of Technology.

Walter Sobkiw is a senior principal engineer with E-Systems, ECI Division. He is currently a member of the Advanced Technology Team and is responsible for defining system development methodologies and new business pursuits. He holds a BSEE from Drexel University.