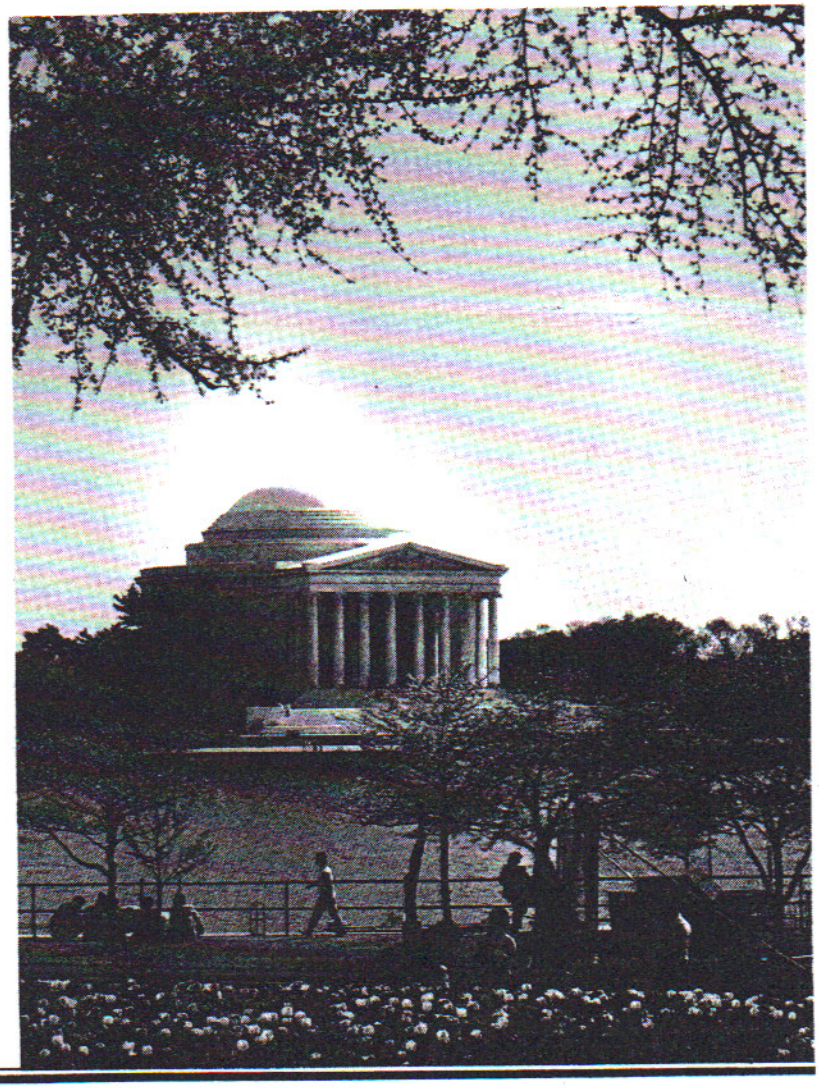
Proceedings of the Sixth Washington Ada Symposium

June 26 - 29, 1989

Sponsored By:
DC Chapter ACM & SIGAda
NASA Goddard Space Flight Center
Federal Aviation Administration
U.S. Army Information Systems Engineering Command

In Cooperation With:
American Institute of Aeronautics and Astronautics



WADAS'89 Washington Ada Symposium

BINDING AS A MECHANISM TO SUPPORT REUSABILITY IN A DISTRIBUTED Ada COMMUNICATIONS SYSTEM

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

ABSTRACT

An effective mechanism for supporting the binding of functions is key to the support of reusable software in a distributed communications system. By adopting an architecture that recognizes the distinctive role of functional objects - which are directly traceable to functional requirements, and binding objects - which serve only to bind the functional object to the operating system services and to the hardware, the portability and reusability of each object is enhanced. This software architectural approach also suggests a set of paradigms for the binding objects. This paper describes a 'novel' binding mechanism to support effective reusability, and will raise issues for realistically achieving the goals of reusability in distributed Ada oriented communications systems.

INTRODUCTION

Reusability and portability have always been a goal for large software intensive systems. This goal was driven by the desire to minimize the effort required to develop software which in many cases supported the same application or minor variations of the same application. A number of solutions have been proposed and attempted with limited success. The present approach for minimizing the long term software investment is based on reusability in which functional software integrated circuits (ICs) are bound to the hardware by binding objects to form different applications. (Software ICs, cox[2], can be defined as components of general functionality, and used in various applications. A goal for software reusability would be to mimic the reusability success of hardware ICs.) In all cases the primary goal is to minimize the software investment. These concepts are extremely complex and involve many aspects of the software development activity.

THE PROBLEM AND PRESENT SOLUTIONS TO REUSABILITY

When software systems are developed, they tend to be viewed in terms of their functional, operating system, and hardware requirements. The operating system and hardware requirements are driven by various commercial offerings, or custom solutions developed inhouse or combinations of commercial and custom operating systems and hardware. Figure 1 illustrates this concept. Methodologies such as Structured Systems Analysis and Object Oriented Design (OOD) are based on this concept. The problem with this view of the software activity is that the critical software component is missing or is not adequately being addressed in the design process. This software element is what effectively binds the software to the operating and hardware elements in the system.

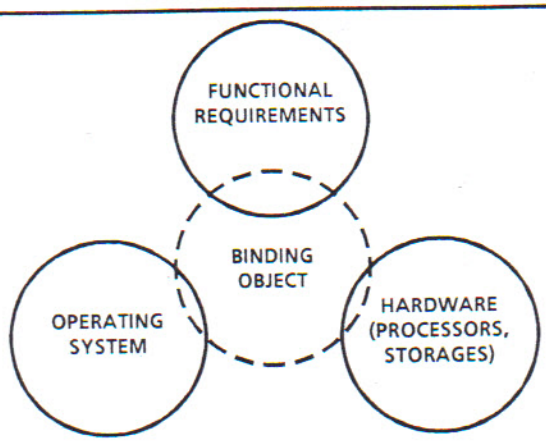


Figure 1. The Software Activity

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

The encapsulation and inheritance advocated by object oriented design are important mechanisms in achieving the goal of software reusability. Cox [2] clearly illustrates these features. What is not discussed in this design method is that the functional software architecture should not be encapsulated with the

hardware architecture or operating system services. This can have a profound impact on the portability and reusability of the application software components. 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. This one time procedure is seldom successful. The allocation of functions 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 [1]. Chen and Steimle [7] 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.

The second assumption was that the software designer does not need to understand the hardware or operating system 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 [3] and Burger [6] 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.

THE SOLUTION TO REUSABILITY VIA BINDING

The proposed novel binding approach design method is different from OOD, which does not distinguish architecture objects from functional objects, and different from structured system analysis and its derivatives, which considers software architecture design a small 'adjustment' after an implementation independent analysis of the requirements. This novel approach is based on two considerations which are not advocated in the 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 the hardware and 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 or the operating system services is not a one time event in the life cycle of a software project. This is especially true in a high performance embedded system. This point was expanded upon in a discussion about fault tolerance, and performance by Chen and Sobkiw [4]. Thus, if an effective mechanism could be developed to isolate unique 'binder object software' from 'functional object 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.

Referring back to Figure 1, 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 2. As shown in Figure 2, the software in a

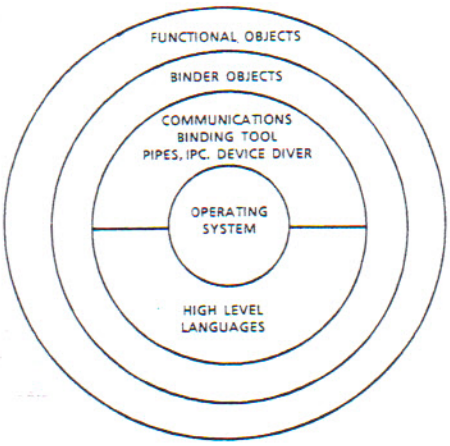


Figure 2. System Layers

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

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, Inter process communications (IPC) mechanisms, and system configuration files that not only translate application program source code to executable code, but also defines the profile of the application and binds 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. This layer supports translation of functions into processes, defines interprocess communication, allocates processes to hardware, and supports backup/ failover operations. The outer layer of this collection of elements are the functional objects. The functional objects must follow the interface rules to the inner layer while satisfying the functional requirements of the application.

A close parallel to this concept is 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 application requirements. For 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.

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. More recently, the spreadsheet and SQL-DBMS has provided a shell for a class of business applications and data base management applications. In the case of a communications system being developed at E-Systems, 1/3 to 1/2 of the software code was responsible for just binding the communications software to the hardware and operating system services. Communications is probably more unique than other applications areas since extensive interaction with the hardware is expected.

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 translates to the architecture of that software IC. The binder objects in Figure 2 must present a standard, well defined, well accepted interface to the functional objects.

THE PROBLEM AND PRESENT SOLUTIONS WITH BINDING

Ada has attempted to address the issues of binding the applications software to the hardware by providing a common architecture for what were once considered languages and operating systems. The design of the Ada language can be seen as an attempt to absorb the binding objects into the language. Ada however can also be implemented as a facade over UNIX. If an Ada application is developed to execute in a UNIX environment the designers will have access to a unique hardware/operating system architecture. In many cases control structures used to support many unique requirements such as fault tolerances are left to the Ada applications software developer and the level of abstraction. The control structure may not be sufficiently high in Ada, as found in the transaction processing paradigm, to support these requirements and a clean consistent interface between the architecture and the functional application software ICs.

There are a variety of traditional software binding mechanisms available in different systems, but what is available in one system is not always available in another system. The conventional primary binding mechanisms today are as follows:

- UNIX PIPE
- Inter Process Communications
- Shared Memory
- Ada Package, Subprograms, Etc.

Each of the above binding mechanisms have unique characteristics. The UNIX PIPE is a fully dynamic binding mechanism in which each component has no information about the connection. The path could be via memory or Ethernet bus. The IPC is a loosely coupled dynamic binding mechanism. The software components do not have to be linked or assembled together during system build time. The components must know each others name and this can be performed by hard coding the name or using logical addressing via command line parameters. Shared memory is a tightly coupled binding mechanism. Just as in the IPC, the components do not have to be linked or assembled together. Each component must know the name of the shared memory and interface synchronization mechanism. This can be either performed by hard code or by command line parameters. Ada is a static binding mechanism. Each component must know the service and interfaces of the connected software component. The binding is completed at build time.

THE NOVEL BINDING APPROACH

The problem with these binding mechanisms include inconsistent implementations between vendors and the lack of supporting a higher level of binding abstraction. It is not totally clear what the requirements should be to support this high level binding abstraction. Potentially the ideal high level binding operators required to combine reusable software objects into useful applications systems should support the following kinds of requirements:

- Allocate software functions into software processes
- Connect different software functions to the same software process
- Connect different software functions to a different software process
- Channel inputs to the proper software function
- Channel outputs from a software function to a proper device
- Allocate processes to hardware resources

Each hardware computer vendor could provide such tools together with the performance data to the applications integrator. The method of connecting software must be established and standardized. This is the foundation of reusability based on the software IC concept.

The need for this function is illustrated by its use in an Ada communications system. In this Ada system, the software functions are allocated to processes which are then allocated to hardware resources. The process talks serially to hardware resources such as I/O devices and executive resources such as device drivers. Roughly 1/2 of the CPU power is used for functional application processing and 1/2 of the CPU power is used for communicating with cooperating processes and I/O devices. The optimum architecture may be different for each system even when the functional application is the same. This is driven by unique requirements such as fault tolerance. As shown in Figures 3 and 4 the arrangement of function and processes is very sensitive to traffic flow and reliability requirements. Requirements such as reliability may specify backup hardware, backup processes, or both. The issue is that the functional software should be independent of these architectures and their characteristics and the effective implementation of these requirements should be part of the binding mechanism.

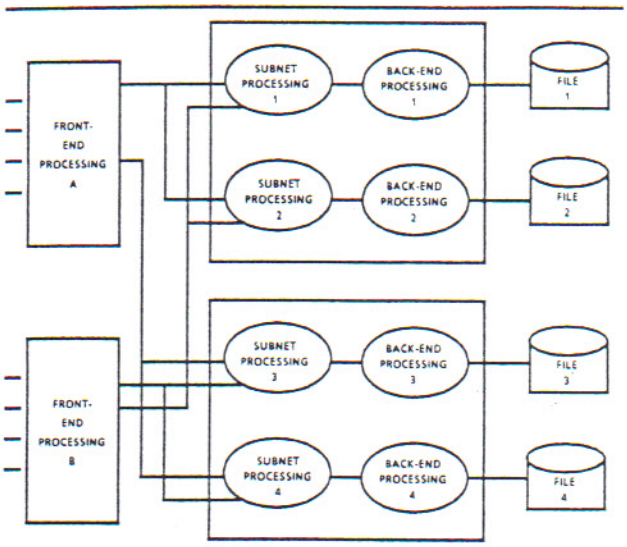


Figure 3. Architecture A

An architecture that favors no cross subnet traffic and low I/O overhead system communications.

The 'novel' binding approach has been applied to a communications system at E-Systems. The intent was to develop a software design based on the assumption that a large portion of the software code would be architecture dependent. As shown in Figure 5, the software design was partitioned into architecture dependent components and functional dependent components. They are as follows:

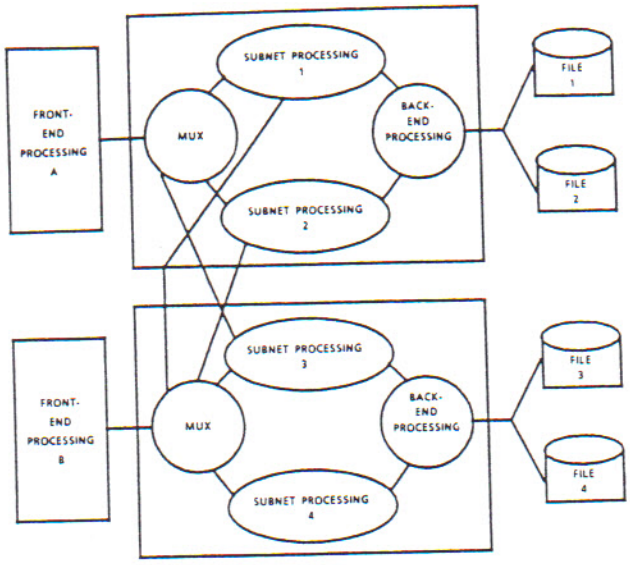


Figure 4. Architecture B

A communication systems architecture that supports blocking of data for high I/O overhead systems.

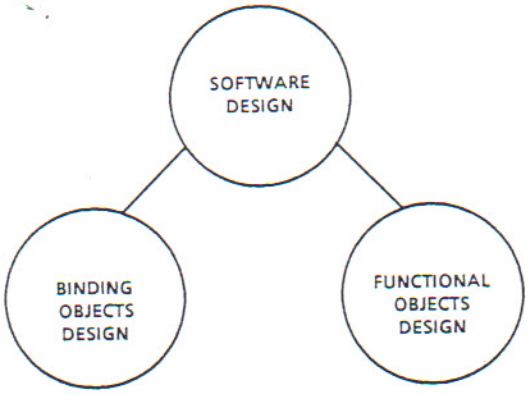


Figure 5. Potential Software Method Supporting Reusability

Software project can be partitioned in two separate design activities.

- Binding Objects
  - Main line procedure for each process
  - System Access Package (SAP)
- Functional Objects

Figure 6 illustrates the Mainline process and Figure 7 illustrates how the SAPs were used to bind the functional Ada applications software to the hardware and UNIX based operating system.

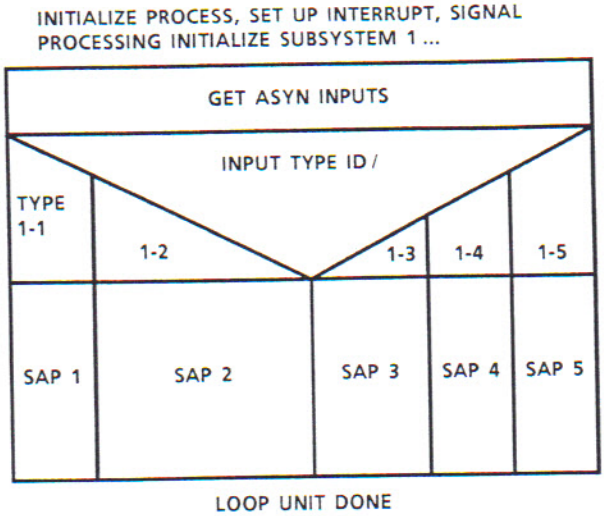


Figure 6. Mainline

The mainline does not contain any functional application requirements.

As shown in Figure 6, the mainline procedure developed with this 'novel' binding approach performs a number of functions. It initializes the SAPs included in a process by calling the initialization service entry provided by each SAP included in the process. It sets up the signal interrupt interface within the operating system and relates the signal to the proper SAP. When the initialization is complete it waits for the signal interrupt. There is no functional application code in the mainline. There is only architecture dependent code as defined by the operating systems and hardware services.

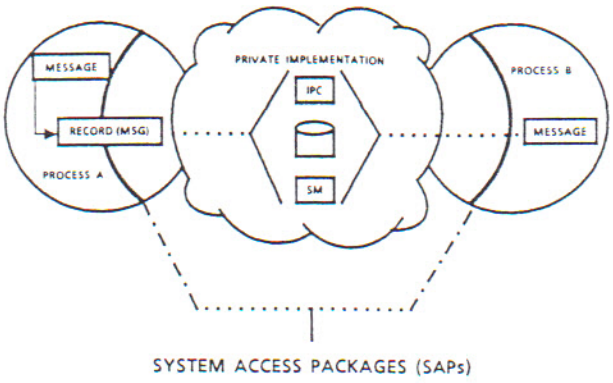


Figure 7. System Access Package (SAP)
The SAP interfaces the functional application to the operating system and hardware environment.

As shown in Figure 7, the SAPs are effectively architecture dependent code paired with other architecture dependent code or paired with I/O devices (device drivers). The SAPs are implemented by one single authority even though each part is linked to a different process. This is a significant departure from convention. The SAPs present themselves to the functional code as an Ada procedure interface. In other words, the socket of this software IC has the characteristic of the Ada procedure call. The internal structure of a SAP can change from one set of hardware and operating system services (architecture) to another set with no visible difference to the cooperating functional software. Changes in the internal implementation of the SAP will not change the function of the application but will have a considerable influence on the cost, throughput, and reliability of the application. There are many SAPs in an application but they can be categorized into a few groups according to the architecture feature employed by the SAP. Some of these features are intraprocessor IPC, interprocessor IPC, and shared memory. SAPs of the same type are differentiated by the data they support.

Figure 8 is a typical mainline procedure in Ada, and Figure 9 is the Ada specification of the two objects of a typical SAP connecting two cooperating functional objects. Each SAP object is used by one unique functional object to invoke the service of another functional object. If the architecture requires that the two cooperating functional objects be in the same process, then the two SAP objects are both included in the same process. The bodies of the two SAP objects are essentially null procedures that translate the entry call from one specification to the other specification call. If the architecture requires that the two cooperating functions reside in different processes, then each object in a SAP is included with its related functional object into the proper process. The bodies of the two SAP objects implement the proper interprocess communication mechanism.

```
procedure MAINLINE is
    FRONT_END_SAP.INIT_SAP;
    BACK_END_SAP.INIT_SAP;
    • •
    • •
    for I in 1 • • NUM_OF_INPUT_FILES loop
        -- setup input signal processing
        FILES_IN(I) : = FD_MAP (I);
    END loop;
    -- wait and read any input
    loop
        FILE_OUT : = IPC_SELECT.CALL
            (FILES = > FILES_IN,
             WAIT = > INPUT TIME);
    for I in 1 • • NUM_OF_INPUT_FILES loop
    -- call the proper input processing procedures
    according to the input file end loop;
end loop;
```

Figure 8. The Mainline Sets Up Interrupt Processing, Waits for Input, and Activates the Proper Function Procedure.

```
package MUX_CLIENT is
     procedure INIT_SAP;
     procedure RECEIVE_SUBNET_MESSAGE;
     procedure RECEIVE_MESSAGE_ACK;
     procedure ....
end MUX      _CLIENT;

package MUX_SERVER is
     procedure INIT_SAP;
     procedure SEND_SUBNET_MESSAGE;
     procedure SEND_MESSAGE_ACK;
     procedure ....
```

Figure 9. Two of the Assosciated Package Specifications in a SAP.

Table 1 summarizes the amount of code found in the architecture dependent code and functional application dependent code for an Ada-based communication system at E-Systems. The mainline and SAPs are system dependent software items. These items are reusable in different applications in the same hardware and operating system environment but are not easily portable to distinctly different systems. The rest of the software is functional application software which is reusable in different applications and portable to different systems.

In the examples illustrated it is clear that the reusability of any code is limited by data type. Most modern high level languages advocate strong type checking. Software developed for one class of data cannot be used to process another class of data even when it is determined that the function is common. However, different classes of data can be adapted to architecture objects and functional objects by facilities like the 'generic' in Ada.

Tue Aug 16 13:23:49 1988 short cnt Page 1

Architecture Software Packages

Subsystem Name	Pack-ages	Lines In File	Blank Lines	Com-ment Lines	Compat-ible Lines	State-ments
Mainlines	72	12,635	1,865	1,824	8,946	6,172
Access specific	51	9,785	877	6,954	1,954	1,249
Access packages	305	20,027	3,603	441	12,008	5,722
Total					22,908	13,143

Functional Software Packages

Subsystem Name	Pack -ages	Lines In File	Blank Lines	Com-ment Lines	Compat-ible Lines	State-ments
Data dictionary	23	13,387	1,080	2,474	9,833	1,764
Common functions	57	10,470	2,059	3,857	4,554	2,804
MMI functions	166	38,227	4,619		14,583	5,379
Front-end processing	97	9,399	1,439	5,243	2,717	1,685
Back-end functions	133	24,921	3,704	4,995	16,222	8,776
Total					47,909	20,408
Architecture/ function					48%	64%

Table I. This table lists the type of code produced for a real time data communication application.

NOTE: In this multimedia communication application, the codes that are dedicated to binding the functional objects to the hardware and operating system are a large portion of the total code produced for the application. These binding codes are unique to each system, but are generic in nature.

CONCLUSIONS

Reusability is an extremely complex issue that involves architecture, requirements, design and tools such as an effective binding mechanism. In the case of a communications system, there is a high probability that 1/3 to 1/2 of the implementation code is machine dependent. Since it is not likely that machine architectures will be standardized in the near future it is reasonable to assume that any reusability effort on a communications program will be only of moderate success. In addition, for reusability to be successful, interfaces between reusable components must be standardized. In other words, an architecture must be defined and accepted for each reusable component. The Ada package and procedure constructs are an important ingredient to support the effective definition of reusable software interfaces. However, the architecture definitions of large numbers of software ICs prior to some initial transition steps may be too high of a goal to achieve at this time. It is more practical to assume that architectures can be more readily defined and accepted for large pieces of software subsystems such as a subsystem that effectively binds computer architecture independent communications functions to each vendors unique hardware architecture.

This paper has defined a 'novel' binding approach to support reusability goals. This binding approach is high level and applicable to several technology areas. This binding mechanism begins by acknowledging that the system development must be partitioned into architecture dependent and functional dependent software entities. An automated tool can and should be provided to facilitate the architecture design which binds the functional design. This tool can be used at the inception of the software project with stubbed functional components all the way through the maintenance phase of the software project when the changes in load scenarios and performance necessitate architecture changes. The binding tool like the printed circuit board and the BUS is the foundation of software IC.

REFERENCES

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

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

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

4. 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.

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

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

7. Thomas L. Chen and Cheryl L. Steimle, "Two Design Approaches Using The Ada Language", IEEE Southeastcon 87, Vol. 1, pg 72.